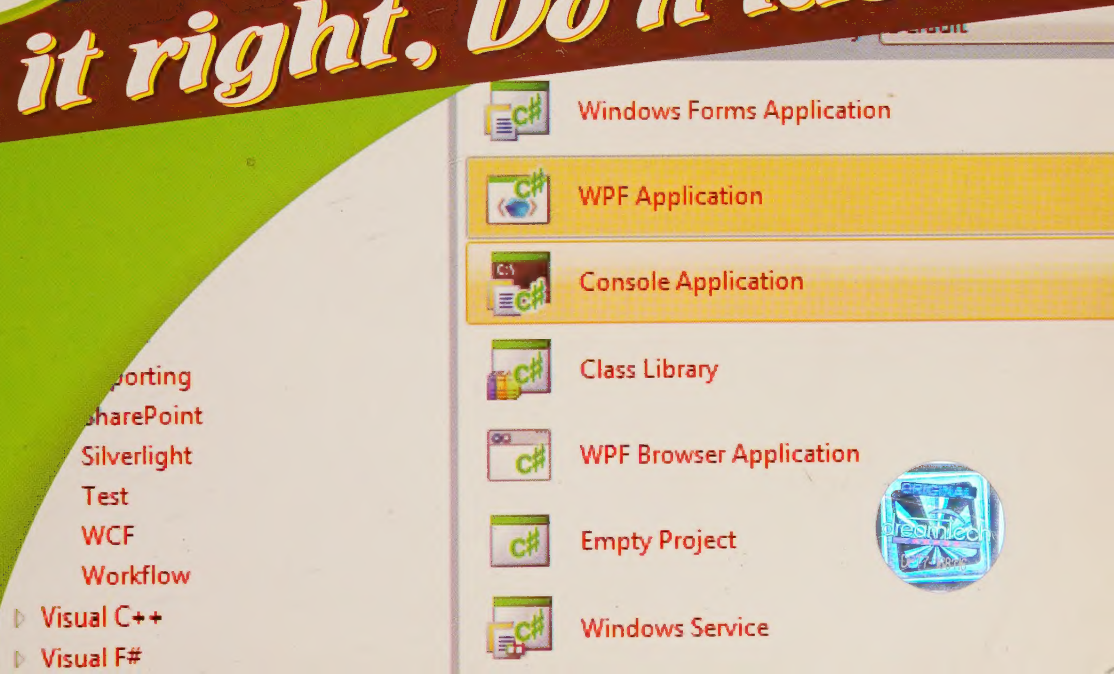
dreamtech PRESS

- An all-inclusive book to teach you everything about C# 2010
- Easy, Effective, and Reliable
- Quick and Easy learning in Simple Steps
- Most preferred choice worldwide for learning C# 2010

C# 2010 IN SIMPLE STEPS

Easy to learn, with hundreds of illustrations

Do it right, Do it fast!



KOGENT Learning Solutions Inc.

C# 2010

IN SIMPLE STEPS

Authored by:

Kogent Learning Solutions Inc.

Published by:

dreamtech PRESS

ISBN: 978-93-5004-032-4

Reprint Edition: 2012

Printed at : Ar Emm International, New Delhi

Table of Contents

Chapter 12 ■ Introduction to Windows Workflow Foundation 327

Chapter 13 ■ Working with Web and WCF Services 345

Chapter 14 ■ Deployment of C# 2010 Applications 361

Chapter 1

Introducing .NET Framework 4.0 and Visual Studio 2010

In this Chapter:

- Describing the Benefits of the.NET Framework
- Explaining the Architecture of.NET Framework 4.0
- Explaining the Components of .NET Framework 4.0
- Exploring the Key Features of Visual Studio 2010
- Installing Visual Studio 2010
- Introducing the Visual Studio 2010 IDE
- Creating Simple Visual Studio 2010 Applications

The.NET Framework is one of the most popular and widely used integrated software development environments today. Before its advent, software developers faced a lot of difficulties while integrating code that was written in different programming languages. This was because each language required a different execution environment to execute the code written in that language. For instance, the code written by using Visual Basic 6.0 required a different execution environment than that required by code written by using Visual C++. Later, Microsoft introduced the .NET Framework, which provided programmers a single platform to develop console, Windows, and Web applications in various programming languages, such as Visual Basic and Visual C#. The .NET Framework helped to reduce the complexities involved in developing large and reliable .NET applications.

Microsoft introduced a new development and code execution software, called Visual Studio .NET, to deliver the features and services offered by the .NET Framework. Visual Studio .NET is a set of tools designed to help application developers build complex .NET applications and create innovative solutions. It provides the necessary environment in which developers can create and execute various types of .NET applications, such as console applications and Windows Presentation Foundation (WPF) applications. Visual Studio .NET has also improved the process of application development and made it easier.

Microsoft released the first version of the .NET Framework (.NET Framework 1.0) and Visual Studio (Visual Studio .NET 2002) in 2002. The second versions of the .NET Framework and Visual Studio were released in 2003 as .NET Framework 1.1 and Visual Studio .NET 2003, respectively. In 2005, Microsoft launched .NET Framework 2.0 and Visual Studio 2005. The .NET Framework 2.0 was considered a major release of the .NET Framework. It was soon followed by the release of .NET Framework 3.0 in 2006. From Visual Studio 2005 onwards, Microsoft removed the .NET alias from Visual Studio .NET. An upgrade of .NET Framework 3.0, .NET Framework 3.5, was released in 2007, along with Visual Studio 2008. .NET Framework 4.0 was released in 2010, with several new features aimed at enhancing the programming experience of the .NET developer. the same year saw the release of Visual Studio 2010. Both Visual Studio 2010 and .NET Framework 4.0 aimed at providing a user-friendly environment to write applications, such as console, Windows, Web, and WPF, as well as Web services.

Before you begin C# programming, you must have a basic understanding of the .NET Framework and the Visual Studio 2010 Integrated Development Environment (IDE).Therefore, the chapter begins with the benefits, architecture, and components of .NET Framework 4.0. It also introduces you to the key features of Visual Studio 2010 and acquaints you with the installation process of Visual Studio 2010. Toward the end of the chapter, you learn to create and run simple console, Windows, and Web-based applications.

Let's begin with discussing the benefits of the .NET Framework.

Describing the Benefits of the .NET Framework

The .NET Framework is an essential Windows component that supports the development and deployment of console, Windows, Web-based applications and services. It is designed to provide a code execution or runtime environment with minimal versioning conflicts and application deployment issues.

Note

Console applications are command-line based applications that accept input and display the output at the command-line. Such type of applications have a very simple user interface (UI) and often require little or no user interaction. Applications that have a Graphical User Interface (GUI) and can be run locally on users' computers are known as Windows applications. These applications can be created by using any of the .NET programming languages, such as C# and Visual Basic, while taking advantage of the .NET Framework debugging tools. Applications that are meant to be hosted through a Web browser over a network, such as the Internet or Intranet, are referred to as Web applications.

The .NET Framework offers a lot of benefits to software developers, such as the following:

- **Comprehensive and consistent programming model:** Provides a consistent object-oriented programming model across various languages. You can use this model to create programs to perform different tasks,

such as connecting to databases and retrieving data from them as well as reading from files and writing to them.

- **Cross-platform support:** Enables interoperability among those Windows operating systems that support Common Language Runtime (CLR). In other words, any Windows operating system that supports CLR can execute a .NET application.

Note

CLR is the runtime environment that forms the foundation of the .NET Framework. It is responsible for managing code execution at runtime and provides services, such as automatic memory management.

- **Cross-language interoperability:** Facilitates the reuse of code as the .NET Framework enables code written in one programming language, such as C# to interact with code written in a different programming language, such as Visual Basic.NET. This helps to improve the efficiency of the overall development process.
- **Automatic resource management:** Manages and frees resources, such as files, memory, and database connections when you no longer need them.
- **Ease of application deployment and maintenance:** Enables .NET applications to be easily deployed. For applications that are not based on the .NET Framework, you simply need to copy the application along with the components it requires, in the directory of the target computer. With the .NET Framework it is possible to quickly install and deploy applications such that the installation of new components does not affect the already existing applications.

After introducing you to the .NET Framework and learning about its benefits, let's discuss the architecture of .NET Framework 4.0.

Explaining the Architecture of .NET Framework 4.0

.NET Framework 2.0, 3.0, and 3.5 (along with their service packs) form the foundation of .NET Framework 4.0. In other words, the architecture of .NET Framework 4.0 includes the components of .NET Framework 2.0, 3.0, and 3.5 along with some additional enhancements, as shown in Fig.C#-1.1:

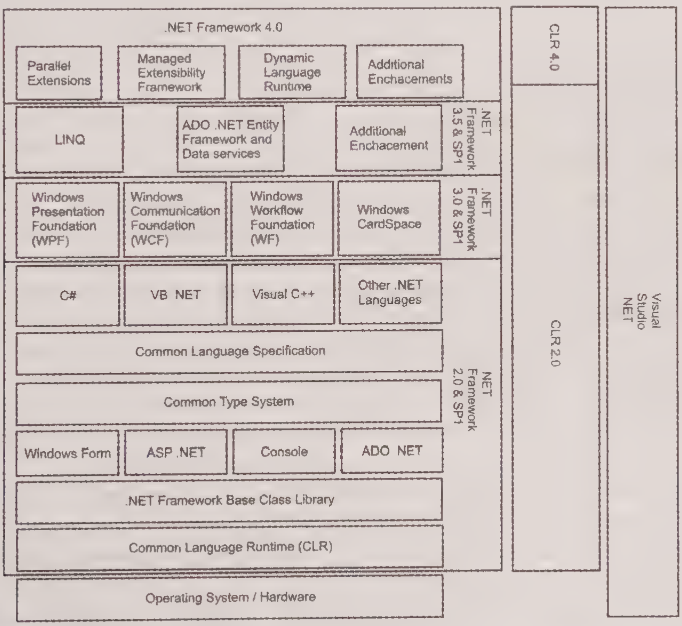


Fig.C#-1.1

As shown in Fig.C#-1.1, .NET Framework 2.0 consists of components, such as.NET Framework Base Class Library, Windows Forms, ASP.NET, Console, ActiveX Data Objects for .NET (ADO.NET), Common Language Specification (CLS), Common Type System (CTS), and .NET languages, such as C# and Visual Basic.NET. Similarly, .NET Framework 3.0 comprises WPF, Windows Communication Foundation (WCF), Windows Workflow Foundation (WF), and Windows CardSpace (WCS). .NET Framework 3.5 introduces a few enhancements, such as Language-Integrated Query (LINQ), ASP.NET AJAX, ADO.NET Entity Framework and Data Services, and new compilers for C#, Visual Basic.NET, and C++. Finally, .NET Framework 4.0 incorporates Parallel Extensions, Managed Extensibility Framework, Dynamic Language Runtime, and other additional enhancements.

Let's now learn about some of the important components of .NET Framework 4.0.

Explaining the Components of .NET Framework 4.0

As mentioned earlier, .NET Framework 4.0 is based on .NET Framework 2.0, 3.0, and 3.5 and therefore includes several components that are a part of these versions of the .NET Framework. Some of the basic components of .NET Framework 4.0 are as follows:

- CLR
- CTS and CLS
- Assemblies and metadata
- .NET Framework Base Class Library
- Windows Forms
- ADO.NET
- ASP.NET and ASP.NET AJAX
- WPF
- WCF
- WF
- WCS
- LINQ

Let's now discuss these components one by one in detail in the following sections.

Common Language Runtime (CLR)

One of the most important and central components of the .NET Framework is CLR. Also referred to as runtime, CLR essentially provides a runtime or execution environment within which all .NET applications run. The functionalities that CLR provides include memory management, exception handling, debugging, security, thread execution, code execution, type safety, verification, and compilation. In simple words, you can say that CLR manages the execution of .NET applications. The code that runs under CLR or targets CLR is known as managed code, while unmanaged code is the code that is compiled into native or machine code and is directly executed by an operating system. As managed code benefits from various features such as cross-language integration, enhanced security, type safety, and memory management, using the managed execution environment enables programmers to commit fewer mistakes, thereby create more secure and stable applications.

Interoperability between various .NET languages, such as Visual C#, Visual Basic, or Visual C++ is facilitated by CLR, as it provides a common environment for the execution of code written in any of these languages. All .NET applications, when run, are compiled into an intermediate code called the Microsoft Intermediate Language (MSIL) or IL code, by the language compiler. This MSIL code is then used by the Just-In-Time (JIT) compiler to compile the code into the native machine code, which is the final executable code. Fig.C#-1.2 illustrates the functioning of CLR:

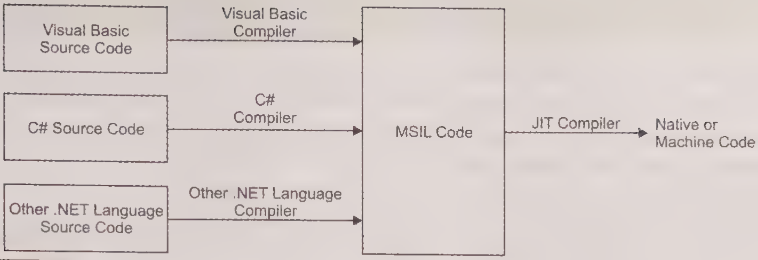


Fig.C#-1.2

CLR is responsible for various aspects of .NET applications during their execution, such as memory management, garbage collection, exception handling, and type safety. In fact, the managed environment of runtime eliminates many common software issues, such as memory leaks and invalid memory references.

Common Type System (CTS) and Common Language Specification (CLS)

An important component of the .NET Framework that provides support for cross-language integration among .NET-enabled programming languages is CTS. CTS defines a common set of types that can be used in applications written in different .NET languages and the operations that can be performed on these types. Two CTS-compliant languages do not require type conversion, when calling a code written in one language from a code written in another language. For all the languages supported by the .NET Framework, CTS provides a base set of data types; however, each language uses aliases for the base data types provided by CTS. For instance, the keyword Integer in Visual Basic and int in C# refer to a 32-bit signed integer. CTS therefore uses the data type System.Int32 to represent a 32-bit integer value.

CLS is a subset of CTS and defines a set of basic language features and programming constructs that all .NET applications have in common and the rules to which they must conform. This enables interoperability between two .NET-compliant languages. The rules defined in CLS serve as guidelines for third-party compilers, designers, and library builders. CLS being a subset of CTS, the languages supported by CLS can use each other's class libraries similar to their own. Application Programming Interfaces (APIs), which are designed by following the rules defined in CLS, can be used by all .NET-compliant languages.

Assemblies and Metadata

An assembly is the basic building block of any .NET application. It is a single, logical deployment unit. Self-describing in nature, assemblies contain all the information that is needed by CLR to execute an application that targets CLR. Every .NET application is compiled into an assembly, which forms a unit of deployment, versioning, code reuse, and security of the application. This implies that everything you do in a .NET application occurs in the scope of an assembly.

An assembly is created in the form of a Portable Executable (PE) file. This PE file can either be a dynamic link library file (.dll file) or an executable file (.exe file) that contains the MSIL code of the compiled application. An assembly is a collection of types and resources that include assembly metadata, and type metadata besides the MSIL. Assembly metadata is the metadata about the assembly itself and its components, while type metadata is the metadata about the types that are required for the assembly, such as classes, interfaces, enumerations, and structures. Two types of assemblies can be created in .NET, private assemblies and shared assemblies. A private assembly is used by a single .NET application and stored in the application's directory while a shared assembly is referenced by more than one application. Shared assemblies must have a unique name (called strong name) and are stored in Global Assembly Cache (GAC).

Metadata describes a program that is in the form of binary information stored in an assembly. When code is compiled in a PE file, the metadata is inserted into one part of the file, while the code is converted into MSIL and inserted into the another part of the file. Metadata describes every type and member of a program. At runtime, CLR loads the metadata into memory and finds information about the code, such as the classes and members specified in a program.

Metadata contains information about the code inside a program in a language-neutral manner. It includes the following information:

- Assembly information, such as the identity of the assembly, which can be a name, version, culture, or public key; the types of assemblies; other referenced assemblies; and security permissions
- Information about types, such as name, visibility, base class, interfaces used, and members (methods, fields, properties, events, and nested types)
- Attribute information, which modifies the types and members of a class

The information contained in the metadata of a .NET application is used by CLR to create objects, access data, and call member functions used in the assembly at run time.

.NET Framework Base Class Library

As its name suggests, .NET Framework Base Class Library is a huge library of reusable classes, interface, and value types that are available to all the .NET programming languages. .NET Framework Base Class Library is object-oriented, language-independent, easy to use, and enhances the productivity of the .NET developer. This class library consists of a hierarchy of namespaces, and each of these namespaces is a logical grouping of classes, structures, interfaces, delegates, and enumerations. For example, the System.Collections namespace contains interfaces and classes that define various collections of objects, such as lists, queues, hash tables, and dictionaries. This namespace can be further broken down into other namespaces, such as System.Collections.ArrayList, which represent the ArrayList type in .NET. The classes, interfaces, structures, and enumerations are collectively referred to as types. These different types in the .NET Framework are arranged in a tree-like hierarchy of namespaces. This arrangement of .NET types in the form of namespaces resolves the issue of name collisions or name clashes that arises when any two types in a code have the same name. The System namespace is the root namespace for some of the most commonly-used types in the .NET Framework. Some of the important classes of the System namespace are Console, Math, Object, String, Array, Enum, and Delegate. Some important structures of the System namespace are Boolean, Byte, Char, Decimal, Single, Double, and Int32.

Windows Forms

Windows Forms are Windows-based GUI applications that users can install and run on their computers. The basic component of a Windows Forms application is a form, which is a visual rectangular area on which either information is displayed to a user or some input from the user is accepted. A variety of controls can be placed inside this form to enhance the UI of the application. The .NET Framework facilitates the development of Windows Forms applications by using a .NET-compliant language. A Windows Forms application is an event-driven application, which means that some action is performed whenever an event, such as a mouse click or pressing of a key in the keyboard, takes place.

ADO.NET

ADO.NET is a Microsoft technology that provides a consistent way to access data sources and databases of all types. It provides access to various data sources, such as Microsoft SQL Server, and data sources exposed through Object Linking and Embedding Data Base (OLE DB) and Extensible Markup Language (XML). ADO.NET can be used to connect to data sources to retrieve, manipulate, and updatethe data. The most important feature of ADO.NET is disconnected data architecture. In this architecture, .NET applications are connected to the databases only till data is being retrieved or modified. The data access operation in ADO.NET is made possible with the help of two components, namely datasets and the data provider. Datasets, which refer to a cached set of database records, follow disconnected data architecture to access or modify data. Therefore, applications do not require connecting to the database to process each record. In addition, all database operations are performed on the dataset instead of the database. The data provider, on the other hand,can be thought of as a bridge that helps to connect an application to a database. Using a data provider, an application can execute commands on a database and retrieve results from thedatabase.

For more information on ADO.NET, please refer to Chapter 8, ADO.NET and Data Binding.

ASP.NET and ASP.NET AJAX

ASP.NET is a part of the Microsoft .NET Framework, which, with its powerful and rich set of features, enables .NET programmers to develop enterprise-level server-based dynamic and interactive Web applications. With

the ASP.NET Web development model, you can add HyperText Markup Language (HTML), Web server, and custom as well as user controls to a Web form. In ASP.NET Web applications, a Web form is the basic UI entity. You can also specify the behavior of a Web server and the customor user controls and add functionalities for these controls in the code of a Web application. An ASP.NET application can be written by using any of the .NET languages, such as Visual Basic, Visual C#, and Visual C++.

AJAX is an acronym for Asynchronous JavaScript and XML and was earlier known as Atlas. It is an extension to ASP.NET, which means that the AJAX functionality can be added to ASP.NET applications. AJAX facilitates the transfer of information between aWeb server and a client in such a way that AJAX-based applications do not require a Web page to be reloaded again while certain modifications are being made to the page. Using AJAX, .NET developers can send only the modified portions of a Web page through asynchronous calls to the Web server. This decreases network traffic and the processing time of a Web page on the Web server. Gmail is an example of a popular AJAX-enabled Web application.

Windows Presentation Foundation (WPF)

WPF is a component of .NET Framework 3.0 and isused to create applications with visually appealing UIs. Formerly known as Avalon, WPF combines two-dimensional (2D) and three-dimensional (3D) graphics, Microsoft Word documents, Portable Document Format (PDF) files, and multimedia into a single framework. It also offers a consistent programming model to develop WPF applications, in which the UI and business logic can be clearly separated. In addition, WPF offers Extensible Application Markup Language (XAML), which is a markup language created by Microsoft. With XAML, both .NET application developers and designers can easily and efficiently work and collaborate on the development process of WPF applications.You can also use WPF to create a wide range of standalone and Web applications.

For more information on WPF, please refer to Chapter 7, Introducing Windows Presentation Foundation and XAML.

Windows Communication Foundation (WCF)

One of the main technologies introduced with .NET Framework 3.0 was WCF, which provides a unified programming model to build service-oriented applications. For example, you can create a service-oriented application in which a service developed in .NET is used by another application that is built on a different platform or programming language. WCF combines and extends the capabilities of distributed systems, .NET remoting, and Web services. All these features help service-oriented applications to communicate with one another, thereby simplifying the development process.

Note

A distributed system has multiple independent interconnected computers that communicate with each other with the help of a computer network. .NET remoting, on other hand, provides a framework that can be used with ASP.NET Web services to develop distributed Web applications. Using .NET remoting, software components located on the same computer, different computers on the same network, or on computers across separate networks can communicate with each other. A Web service is a method that can be invoked through the Internet and then be used to return a value (if any) to the code invoking the Web service. You learn about Web services in detail in Chapter 13, Working with Web and WCF Services.

Next, let's discuss WF.

Windows Workflow Foundation (WF)

WF is a Microsoft technology introduced in .NET Framework 3.0, which helps define, execute, and manage workflows. A workflow generally refers to a sequence of tasks that are performed to complete a given procedure or produce some result.WF applications consist of a series of shapes and icons that represent some actions. Before the advent of WF, application developers used to write both the business logic and its implementation in .NET languages, such as C#, Visual Basic .NET. However, with WF, the business logic code can be defined in a workflow application while the actual implementation code is written in a .NET language. Therefore, WF provides a declarative programming model to build workflow applications that only has the business logic code. For more information on WF, please refer to Chapter 12, Introduction to Windows Workflow Foundation.

Windows CardSpace (WCS)

WCS is a client software introduced by Microsoft that enables users to use their digital identities over the Internet in a familiar, simple, and secure way. WCS can be thought of as an online virtual information card or an ID card for the Internet, which can be used to validate a user's identity. Using WCS, .NET programmers can build websites and software that are prone to identity related attacks, such as phishing. WCS takes care of the problems of traditional online security mechanisms by reducing the dependence on user names and passwords. WCS uses a separate desktop and cryptographically strong authentication to ensure secure online transactions.

Language-Integrated Query (LINQ)

LINQ is a .NET Framework component that adds native data querying capabilities to .NET languages. It has syntax similar to Structured Query Language SQL, which means that you can use LINQ to write statements in any .NET language, such as Visual C# and Visual Basic.NET. LINQ offers a consistent model to work with data across various data sources and formats, such as SQL Server databases, datasets, objects in a collection, and XML documents. The operations performed by a LINQ query typically consist of three main actions: obtaining a data source, creating the query, and executing the query.

After discussing the components of .NET Framework 4.0, let's now learn about the features of Visual Studio 2010.

Exploring the Key Features of Visual Studio 2010

Visual Studio is an IDE from Microsoft that contains a complete set of development tools for building .NET applications. Visual Studio offers a rich set of tools and features that not only enable the application developers to write and modify their programs, but also helps detect and correct errors. The Visual Studio IDE includes a Code Editor, which is a window where you can write the source code for an application that you are developing. An integrated debugger is also available in Visual Studio, which is a tool that helps to monitor the behavior of an application while it is executing. If there is any error in the application, you can use the integrated debugger to determine the exact location of an error. The integrated debugger provides support for IntelliSense, which is a useful feature of Visual Studio and helps to increase the productivity of a .NET application developer. The IntelliSense feature helps in the development of applications as it automatically generates code in the Code Editor. In addition,Visual Studio provides templates for creating applications, such as Class Library, Windows Control Library, and Web Control Library. Templates to create console, Windows, and Web applications, and Web services are also available in Visual Studio 2010.

Visual Studio 2010 is the latest version of Visual Studio that targets .NET Framework 4.0. It helps to minimize the development time of .NET applications by providing different tools to integrate designers into the overall development process.

Visual Studio 2010 also introduces a multi—paradigm programming language called Visual F#. This language combines the features of functional programming languages, such as C; dynamic programming languages, such as Ruby and Python; and object-oriented programming languages, such as Java and C#.

Note

A functional programming language is one that focuses more on what a program should do rather than specifying how it should do. A dynamic programming language, also referred to as a weakly typed programming language, enables you to declare variables without defining their data types. The resolution of the actual data type of any variable declared in a dynamic language is deferred until run time. Any programming language that is centered on the concept of objects and follows the principles of abstraction, encapsulation, polymorphism, and inheritance, is known as an object-oriented programming language. You learn about the object-oriented programming language in Chapter 4, Introducing Object-Oriented programming Constructs.

Before you start working with Visual Studio 2010, you need to know something about its main features and enhancements. The following is a list of the main features and enhancements introduced in Visual Studio 2010:

- New user interface
- New Extension Manager
- Unified Modeling Language (UML)modeling
- Visualization tools to track changes
- Enhanced support for multi-targeting
- The Editor zoom feature
- The Highlighting references feature
- The Navigate To feature
- The Box selection feature
- Call hierarchy of methods
- Facilitation of faster code generation
- The Consume-First mode of IntelliSense
- Export or import breakpoints
- Dotfuscator changes

Now, let's learn about these features one by one, starting with the new user interface of Visual Studio 2010.

New User Interface

The all-new UI of Visual Studio 2010 has been built on WPF and is designed so that it not only appeals visually but also makes the job of .NET application developers easy. Visual Studio 2010 has a reorganized layout wherein file menus and commands appear on a shelf at the top of the IDE. Fig.C#-1.3 shows the new UI of Visual Studio 2010:

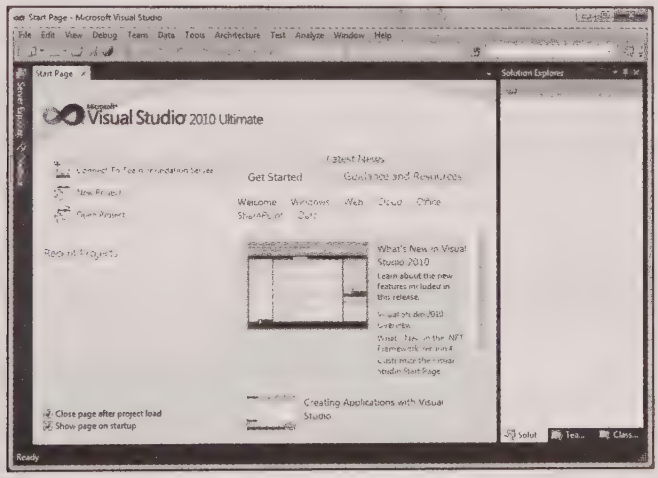


Fig.C#-1.3

Next, let's discuss about the Extension Manager feature introduced in Visual Studio 2010.

New Extension Manager

The Extension Manager feature helps application developers to easily add, remove, enable, and disable Visual Studio extensions. You can download and install these extensions from Microsoft's website, http://visualstudiogallery.msdn.microsoft.com/en-us/site/search?pageIndex=1, as well as from third parties. These extensions include project templates and new controls for ASP.NET, WPF, and Silverlight. You can open the Extension Manager window by selecting the Extension Manager option in the Tools menu. Fig.C#-1.4 shows the UI of Extension Manager:

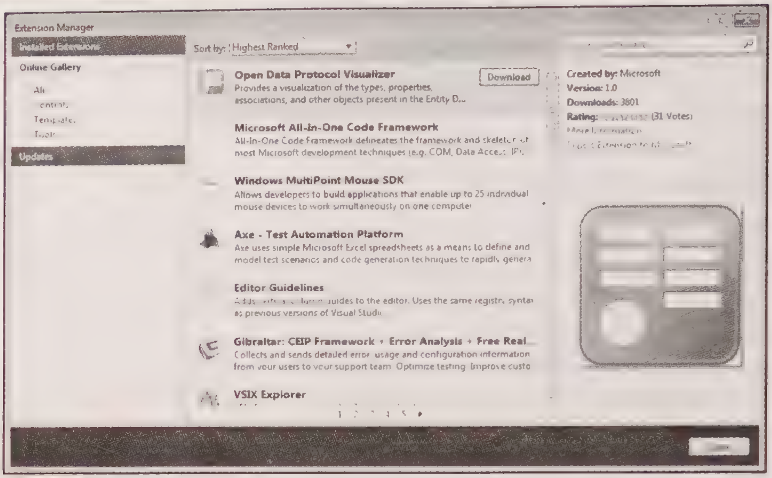


Fig.C#-1.4

Next, let's discuss about the UML Modeling feature of Visual Studio 2010.

UML Modeling

Visual Studio 2010 now supports creation of UML diagrams by introducing UML modeling projects. A UML diagram enables software developers and users to view and understand a software system from a different perspective and in varying degrees of abstraction.

UML modeling projects are expected to facilitate both technical and non-technical users to create and use UML models. The Visual Studio 2010 Ultimate edition provides templates for creating activity, class, component, sequence, and use case UML diagrams. In addition, you can create layer diagrams in Visual Studio 2010 Ultimate edition, which help you define the structure of your system.

You can create a new UML modeling project in Visual Studio 2010 by using the Modeling Project template. You can access this template by selecting the Modeling Projects option from the New Project dialog box, as shown in Fig.C#-1.5:

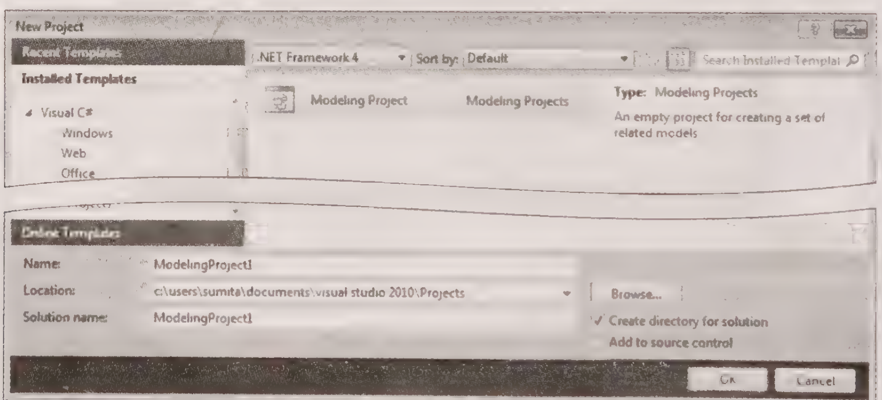


Fig.C#-1.5

When you create a modeling project, it is initially empty, as shown in Fig.C#-1.6:

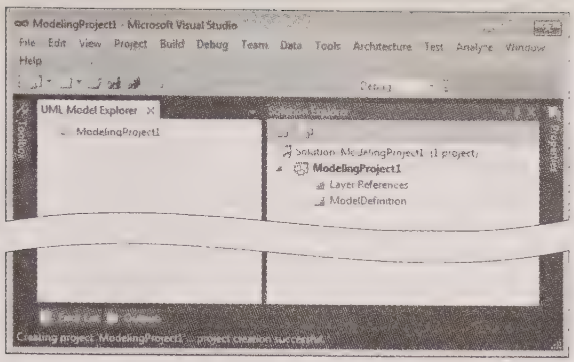


Fig.C#-1.6

You can later add a UML activity, class, component, sequence, or use case diagrams by simply right-clicking the name of your application in Solution Explorer and then selecting the **Add→New Item** option from the context menu. The **Add New Item** dialog box opens, as shown in Fig.C#-1.7:

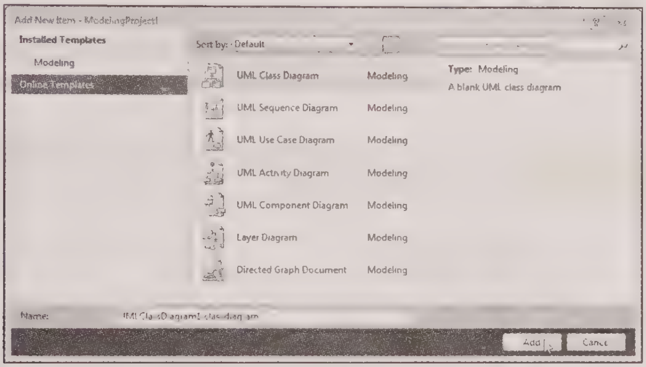


Fig.C#-1.7

As shown in Fig.C#-1.7, various modeling templates are provided, which you can use in your modeling project.

Next, let's discuss the visualization tools that help to track any changes made to the code in the Visual Studio 2010 IDE.

Visualization Tools to Track Changes

Visual Studio 2010 provides a feature known as the visual indicator to keep track of all the changes made in the code of your application. The visual indicator can be seen on the left hand side of Code Editor and it tracks the region or area where you have edited and saved the code. When the visual indicator is yellow in color, it means that the changes done to the code have not yet been saved. When color of the indicator is green, it indicates that the modifications done to the code have been saved. Fig.C#-1.8 shows the visual indicators that can be used to track changes in Code Editor:

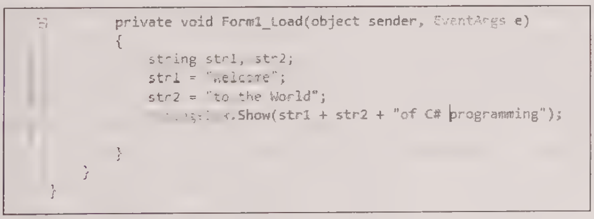


Fig.C#-1.8

Now, let's discuss about the support for targeting multiple versions of .NET Framework provided in Visual Studio 2010.

Enhanced Support for Multi-Targeting

The multi-targeting feature was introduced in Visual Studio 2008 and has been enhanced in Visual Studio 2010.With this feature, you can build .NET applications that target .NET Framework versions 2.0, 3.0, 3.5, or 4.0. In other words, applications developed in older versions of the .NET Framework can be upgraded to .NET Framework 4.0 by using the multi-targeting feature. To facilitate multi-targeting, Microsoft has made available reference assemblies for each version of .NET. When you target a particular version of the .NET Framework, the IntelliSense feature includes only those features and properties that are supported in the selected version of the .NET Framework.

Note

Starting with .NET Framework 3.0, Microsoft introduced a reference assemblies folder that can be found inside the Program Files folder. The reference assemblies folder contains those assemblies that are shipped with the .NET Framework 3.0. These assemblies can be referenced while designing and building new .NET components.

You can target any particular .NET Framework version in the New Project dialog box of Visual Studio 2010 by selecting the .NET Framework version that you want from the drop-down list in the middle pane, as shown in Fig.C#-1.9:

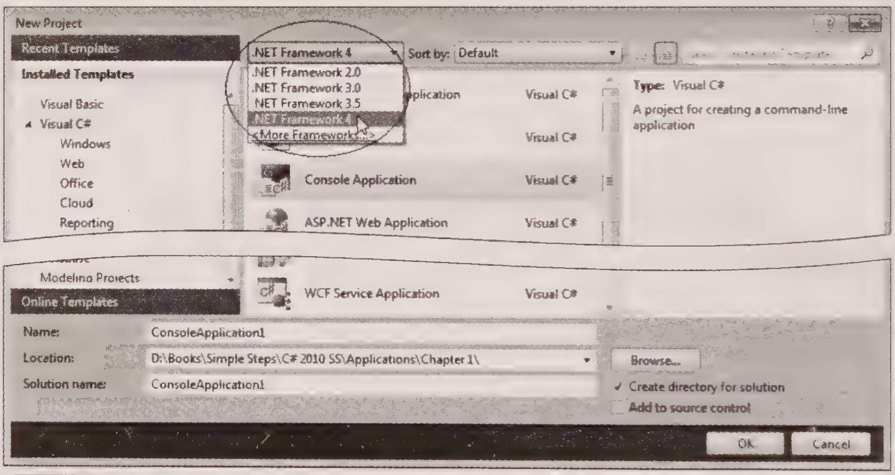


Fig.C#-1.9

The Editor Zoom Feature

With Visual Studio 2010, you can zoom in and zoom out the font size of text in the Code Editor window, similar to the way you do in Internet Explorer or any other Web browser. For this, you just need to hold down the CTRL key on the keyboard and scroll the mouse wheel to increase or decrease the font size of the text in the Code Editor window. An example where the zooming feature can be usefully employed is while giving presentations. In versions earlier to Visual Studio 2010, you could change the font size of the text in the Code Editor window by selecting the Options submenu under the Tools menu. Fig.C#-1.10 shows the increased size of font of the text in Code Editor by using the zooming feature:

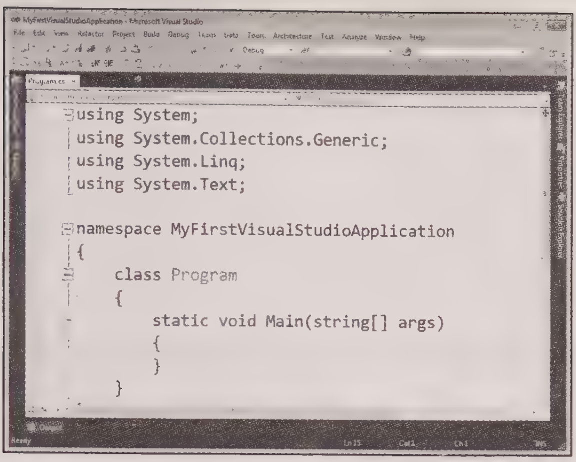


Fig.C#-1.10

Next, let's discuss about the highlighting references feature.

The Highlighting References Feature

Another small yet significant feature added to the Visual Studio 2010 IDE is the Highlighting References feature. Using this feature, all the instances of a class, object, variable, method, or property are highlighted by either simply clicking them or placing the mouse pointer over them in Code Editor. You can also navigate between these instances by using the **CTRL+SHIFT+DOWN** or **CTRL+SHIFT+UP** arrow keys in combination. Fig.C#-1.11 shows the Highlighting References feature in Code Editor:

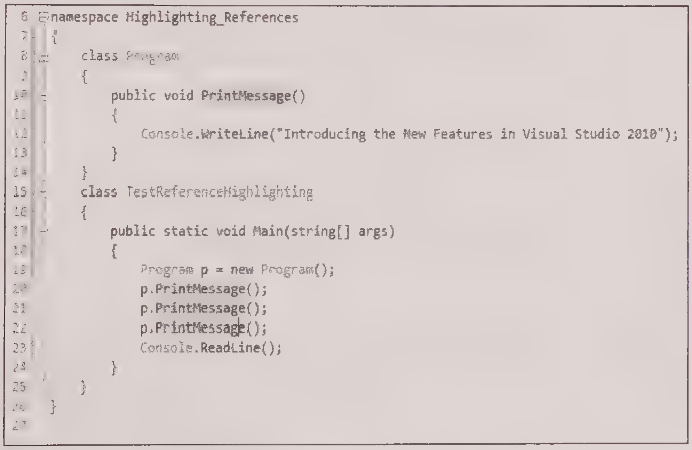


Fig.C#-1.11

As shown in Fig.C#-1.11, all the instances of occurrence of the method, **PrintMessage()** are highlighted.

Note

The Highlighting references feature cannot be used across multiple files currently. This means that it is not possible to select a method in one file and see it highlighted in another open file or files. In addition, references to a symbol, such as a class, object, variable, method, or property cannot be highlighted across assemblies as well. This means that if you select a method call in the code of one assembly, the method is not highlighted in another file from a different assembly.

Next, let's discuss the Navigate To feature of Visual Studio 2010.

The Navigate To Feature

Navigating to a specific symbol or file in the source code has become easier in Visual Studio 2010 with the Navigate To feature. You can use this feature to search for a symbol, variable, word, phrase, or file in your source code. This incremental search feature is especially beneficial when you are not sure of what you are looking for in the source code. For instance, when you enter a symbol, variable, word, or phrase in the Search terms text box of the Navigate To dialog box, the Navigate To feature immediately starts displaying the results, based on the term you enter as you start typing in the Search terms text box. You can also use Camel casing (words beginning in lowercase) and underscore characters in the Search terms text box to divide any phrase that you are typing into keywords. Fig.C#-1.12 shows the use of the Navigate To feature:

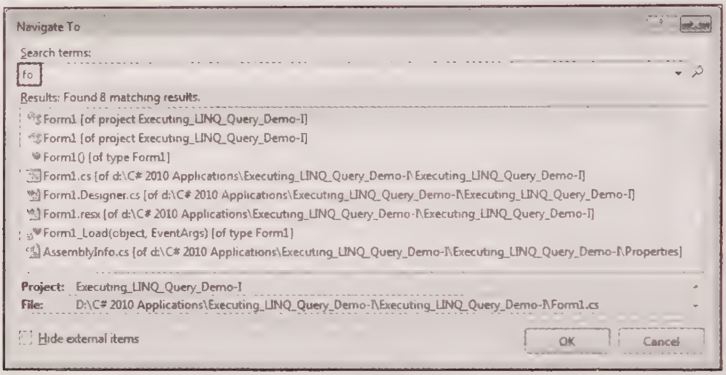


Fig.C#-1.12

As shown in Fig.C#-1.12, when you type the term **fo** in the **Search terms** text box, the results returned in the **Results** section of the **NavigateTo** dialog box are all the file names, method and property names, and event names that contain the term **fo**.

Tip

You can open the NavigateTo dialog box by pressing the CTRL and , keys in combination.

Next, let's discuss about the Box Selection feature in Visual Studio 2010.

The Box Selection Feature

Consider a scenario where you have declared certain variables and methods as public and now you need to modify them from public to private. In such a case, you can go through all the access modifiers one by one and change every occurrence of the public access modifier into private access modifier. However, this can be a tedious work. The new Box Selection feature saves you from such situations. Using the Box Selection feature, you can quickly select a block of text that you want to replace, by pressing the **SHIFT+ALT** keys in combination, as shown in Fig.C#-1.13:

```
public int i;
public int i2;
public string s;
public string s2;
public bool b;
```

Fig.C#-1.13

As shown in Fig.C#-1.13, the selected text appears in a box. After selecting the text, you can enter the new text within the box, as shown in Fig.C#-1.14:

```
private int i;
private int i2;
private string s;
private string s2;
private bool b;
```

Fig.C#-1.14

As you can see in Fig.C#-1.14, typing private once replaces all the instances of public in each of the line. Next, let's discuss about the call hierarchy feature.

Call Hierarchy of Methods

One of the most significant and helpful features added to Visual Studio 2010 is Call Hierarchy. This feature allows you display the following:

- Calls made to and from a selected method, property, or constructor in the Code Editor window
- Implementations of an interface member
- Overrides of a virtual or abstract member

Note

You learn about interfaces and their implementations, abstract and virtual classes, and members in detail in Chapter 4, Introducing Object-Oriented Programming Constructs

You can access the Call Hierarchy window by selecting and then right-clicking a method, property or constructor name in Code Editor. You can then select the View Call Hierarchy option from the context menu that appears, as shown in Fig.C#-1.15:

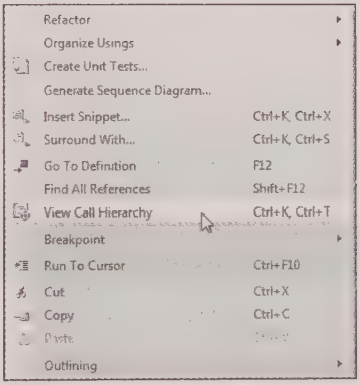


Fig.C#-1.15

In our case, the **Call Hierarchy** window in Code Editor displays the calls that have been made to and from the **PrintInfo()** method, as shown in Fig.C#-1.16:

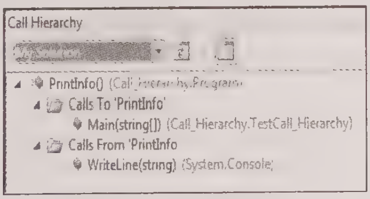


Fig.C#-1.16

The Call Hierarchy feature enables you to navigate through the source code. It also helps understand the flow of code in an application.

Next, let's discuss the faster code generation feature in Visual Studio 2010.

Facilitation of Faster Code Generation

In the Visual Studio 2010 IDE, Microsoft has introduced an interesting new feature called Faster Code Generation. You can understand this feature better through an example. Let's suppose that we have a class called Shapes in an application.

Note

A class can be created in C# by using the class keyword. You can declare and then define variables, methods, and objects inside the class. You learn more about classes and how to create a class in C# in detail in Chapter 4, Introducing Object-Oriented Programming Constructs.

Now, if you write a method or class name that you have not already defined in the application, an error is generated. In our case, we have written a class name Circle, which we have not defined. Fig.C#-1.17 shows a wavy red line that appears below the class name Circle, indicating an error:

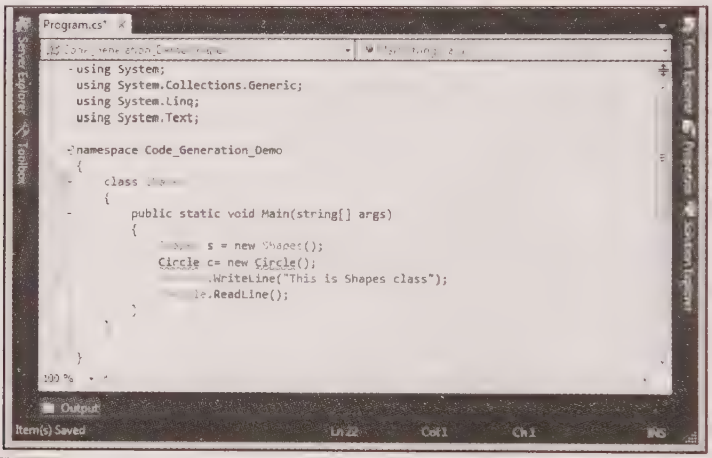


Fig.C#-1.17

Fig.C#-1.17 shows a squiggled line below the class name, Circle. Place the mouse pointer on the class name to display a down arrow below the name. Now, click the down arrow and select the Generate new type option from the drop-down list that appears, as shown in Fig.C#-1.18:

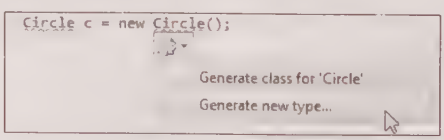


Fig.C#-1.18

When you select the Generate new type option from the drop-down list, the Generate New Type dialog box appears, as shown in Fig.C#-1.19:

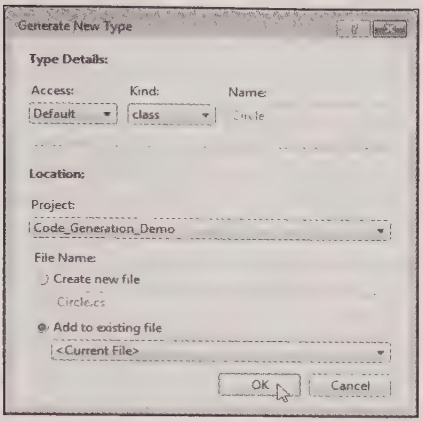


Fig.C#-1.19

Note that in Fig.C#-1.19, we have selected the Add to existing file radio button so that the Circle class can be created in the same application in which the Shapes class is defined. Fig.C#-1.20 shows the code that is generated for the Circle class:

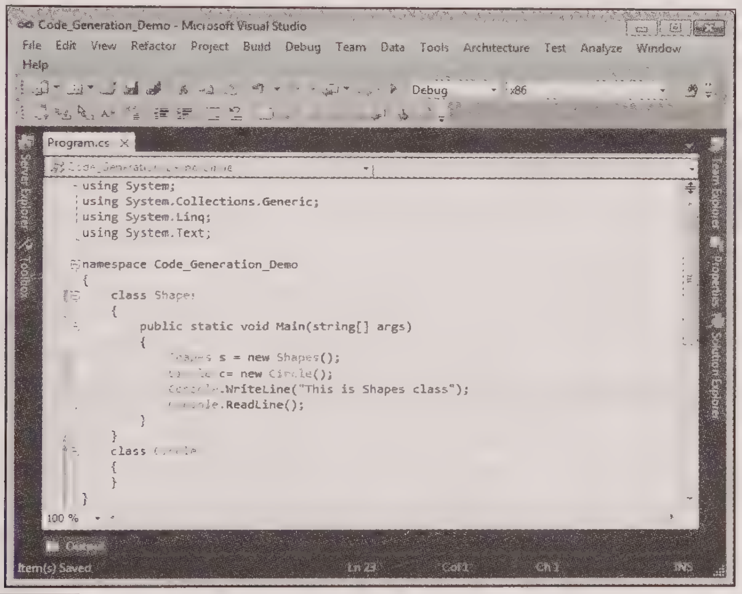


Fig.C#-1.20

Next, let's learn about another new feature of Visual Studio 2010, that is, Consume-first mode of IntelliSense.

The Consume-First Mode of IntelliSense

IntelliSense in Visual Studio 2010 provides two alternatives to complete a statement in the Code Editor window. These alternatives are the suggestion mode and the consume-first mode of IntelliSense. The suggestion mode feature of IntelliSense helps in automatically completing the text as you type in the Code Editor window. To enable the suggestion mode of IntelliSense, you can press the **CTRL+ALT+Space** keys in combination. The consume-first mode of IntelliSense, on the other hand, disables the suggestion mode, which stops IntelliSense from automatically completing the text as you type it. You can disable the suggestion mode of IntelliSense, by pressing the **CTRL+ALT+Space** keys in combination. To enable the suggestion mode of IntelliSense, you can press the same keys again.

Export or Import Breakpoints

Visual Studio 2010 allows you to export or import breakpoints. A breakpoint is a debugging tool that can be used to halt the execution of a .NET application. Breakpoints can be inserted on either a single line of code in a .NET application or on multiple lines of code. The export breakpoints feature allows you to export breakpoints to some other location, such as a different system, to debug a portion of a .NET application. After debugging, the breakpoints can be imported back to the application.

You can export or import breakpoints by using the Breakpoints window, which you can open by selecting Debug→Windows→Breakpoints from the menu bar, as shown in Fig.C#-1.21:

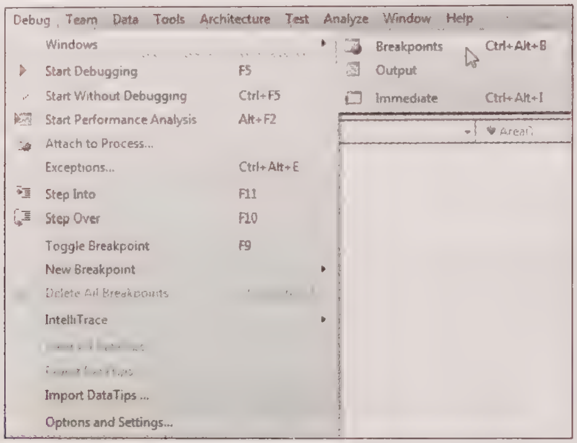


Fig.C#-1.21

This opens the Breakpoints window.

Tip

You can also open the Breakpoints window by using the CTRL+ALT+B keys in combination.

You can insert a breakpoint into a particular piece of code by right-clicking the code and selecting Breakpoint→Insert Breakpoint from the context menu. In Visual Studio 2010, once you have inserted a breakpoint in your application, you can send it to your colleagues along with a bug report. You can pass on breakpoint(s) to your colleagues by using the Export breakpoints button of the Breakpoints window. Breakpoints are exported as an XML file to a location of your choice, from where they can be later imported to a .NET application using the Import breakpoints button of the Breakpoints window, as shown in Fig.C#-1.22:

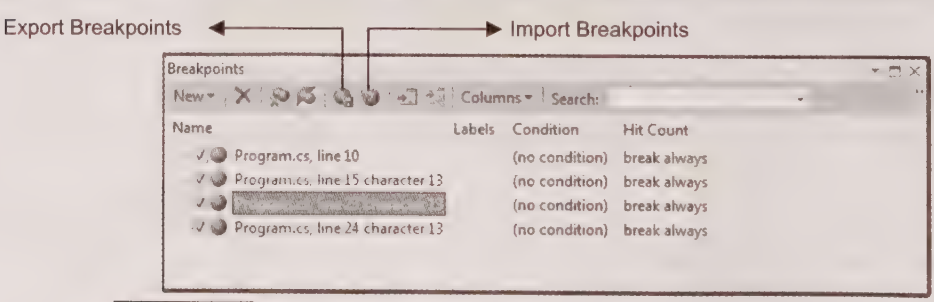


Fig.C#-1.22

This feature is useful when you need to send the breakpoints through electronic mail (e-mail) to another user or want to change all of the breakpoints at once with a text editor.

Dotfuscator Changes

Dotfuscator is a tool that protects, measures, and manages .NET applications. You can find the Dotfuscator tool in the Tools menu in the menu bar of Visual Studio 2010 with the name **Dotfuscator Software Services**. In Visual Studio 2010, this tool includes the Runtime Intelligence functionality and other noteworthy features that include the following:

- **Session Tracking:** Determines what applications have been executed as well as their versions
- **Feature Usage Tracking**: Determines the features being used, the sequence in which they are used, and their duration, in a .NET application
- **Application Expiry:** Encodes an end-of-life date, transmits alerts when applications are executed post their expiry date, and terminates expired application sessions
- **Tamper Defense:** Detects if applications have been modified or tampered

Note

The technologies, managed services and practices used for the collection, integration, analysis, monitoring, and presentation of the usage levels of .NET applications, are collectively known as runtime intelligence.

Now that you are familiar with the key features of Visual Studio 2010, let's learn to install Visual Studio 2010 on your computer.

Installing Visual Studio 2010

Before developing a .NET application, you need to have the required software installed on your computer, such as Visual Studio 2010. However, before you start the installation process, you should take care of several issues, such as system requirements in advance to minimize the chances of encountering problems in the installation. After completing the installation process, you can register the product and install the optional components.

In this section of the chapter, you learn about the system requirements to install Visual Studio 2010, the different editions of Visual Studio 2010, and the installation process of Visual Studio 2010.

System Requirements

You must ensure that your computer has certain hardware and software before installing Visual Studio 2010. Table 1.1 lists the minimum system requirements that your computer should meet to install Visual Studio 2010:

Table 1.1: System Requirements to Install Visual Studio 2010

Hardware and Software	Requirements
Processor	Minimum:1.6 GHz Pentium processor
Supported architecture	×86 and ×64 Windows on Windows (WOW) mode
Operating system	Windows Server 2003 Service Pack 2or above Windows Server 2008 Windows Server 2008R2 or above Windows XP Service Pack 3or above Windows Vista Service Pack 2 or above Windows Server 2008 Service Pack 2 or above Windows 7 (Recommended)
RAM	1 GB RAM or more
Hard disk	For full installation: 3 GB of available hard disk space
Hard disk space	5400 revolutions per minute (RPM) Recommended:7200 RPM or higher
DVD-ROM drive	Required
Display	Minimum:1024 X 768 display
Mouse	Mouse or compatible device

Now, before we show you how to install Visual Studio 2010 on your computer, let's first briefly describe the different editions of Visual Studio 2010.

Editions of Visual Studio 2010

There are diferent editions of Visual Studio 2010 available in the market. Each edition has its own set of features to cater to different users. The following is a brief description of the different editions of Visual Studio 2010:

- **Visual Studio 2010 Ultimate edition:** Provides an integrated environment of tools and infrastructure for the server, which helps to simplify the entire application development process. This edition of Visual Studio 2010 helps produce meaningful business results by using productive, predictable, and customizable processes. It lets you targeting more platforms and technologies, such as cloud and parallel computing.
- **Visual Studio 2010 Premium edition:** Offers an environment that simplifies application development. This edition contains advanced tools to solve the most difficult problems. The advanced code analysis, testing, and debugging tools help individuals and teams to develop high-end applications in which there is a minimal chance of creating bugs.
- **Visual Studio 2010 Professional edition:** Simplifies the basic tasks of creating, debugging, and deploying applications. It enhances creativity with powerful design surfaces. This edition targets several platforms, such as Silverlight, SharePoint, and Cloud applications. The integrated support in Visual Studio for Test Driven Development (TDD) and new debugging tools ensures high quality solutions.

Now let's learn to install Visual Studio 2010 on your computer.

Let's perform the following steps to install Visual Studio 2010:

1. *Insert* the DVD-ROM of Visual Studio 2010 in the DVD-ROM drive of your computer.

   The **Microsoft Visual Studio 2010 Setup** wizard appears (Fig.C#-1.23).

2. *Click* the **Install Microsoft Visual Studio 2010** link, as shown in Fig.C#-1.23:

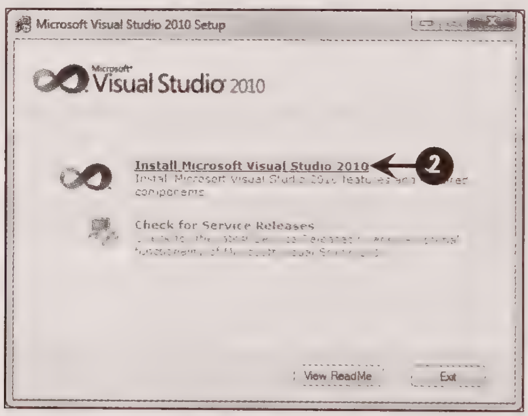


Fig.C#-1.23

A message box appears, which states that the setup is copying the resources required to install Visual Studio 2010 in a temp directory, as shown in Fig.C#-1.24:

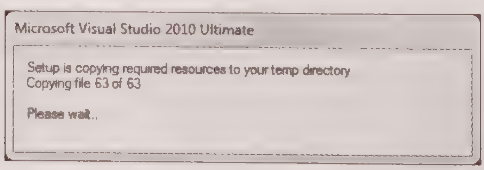


Fig.C#-1.24

When the process of copying the resources finishes, the **Microsoft Visual Studio 2010 Ultimate** page of the installation wizard appears, stating that the setup is loading the components required for the installation, as shown in Fig.C#-1.25:

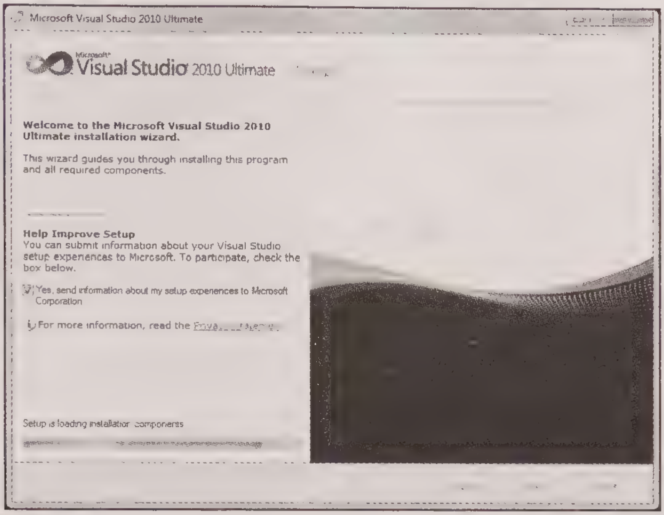


Fig.C#-1.25

After the installation of components, **the Welcome to the Microsoft Visual Studio 2010 Ultimate installation wizard** page appears (Fig.C#-1.26).

3 *Click* the **Next** button after the components are loaded, as shown in Fig.C#-1.26:

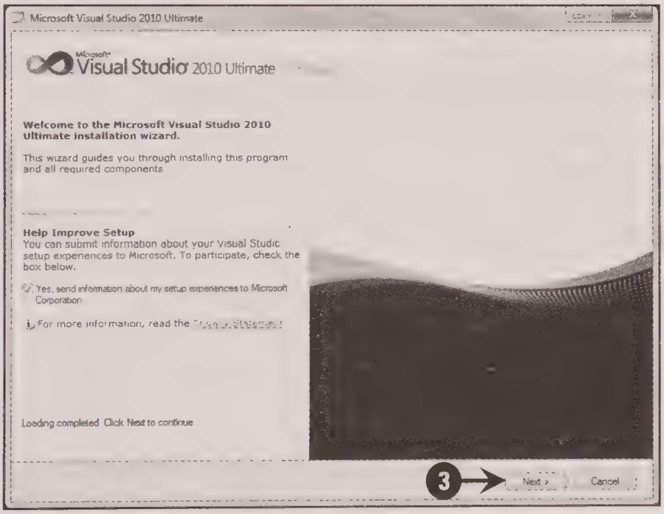


Fig.C#-1.26

The Microsoft Visual Studio 2010 Ultimate Setup– Start Page appears (Fig.C#-1.27).

4 *Select* the **I have read and accept the license terms** radio button after reading the terms of the license (Fig.C#-1.27).

5 *Click* the **Next** button to continue, as shown in Fig.C#-1.27:

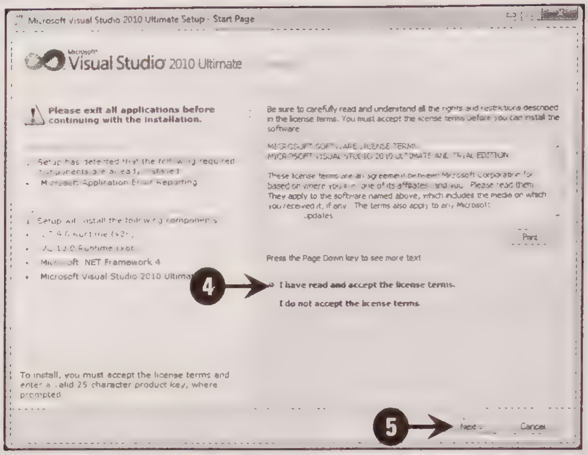


Fig.C#-1.27

The Microsoft Visual Studio 2010 Ultimate Setup – Options Page appears (Fig.C#-1.28). On the left pane of the Microsoft Visual Studio 2010 Ultimate Setup – Options Page, you are presented with a choice of options for installation:

- **Full:** Installs the full version of Visual Studio 2010
- **Custom:** Installs the user selected components of Visual Studio 2010

The right pane of **Microsoft Visual Studio 2010 Ultimate Setup – Options Page** displays the required and available space in the hard disks of your computer as well as the path to install Visual Studio 2010 (Fig.C#-1.28).

6 *Select* the **Full** radio button (Fig.C#-1.28).

7 *Click* the **Install** button after selecting the drive where you want to install Visual Studio 2010. In this case, we are installing Visual Studio 2010 in the default location, as shown in Fig.C#-1.28:

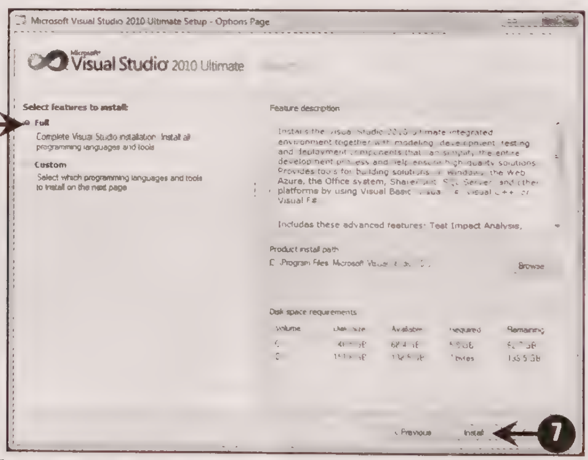


Fig.C#-1.28

The **Microsoft Visual Studio 2010 Ultimate Setup – Install Page** appears, which displays the installing components, as shown in Fig.C#-1.29:

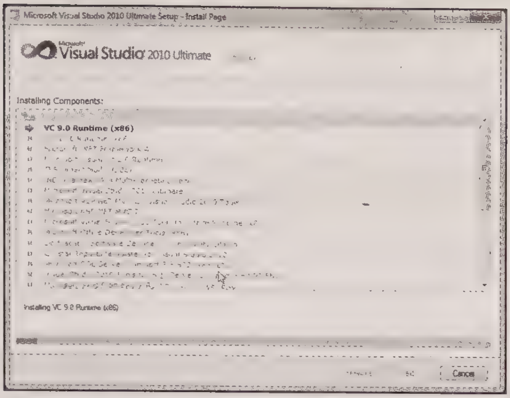


Fig.C#-1.29

8 *Click* the **Finish** button on the **Microsoft Visual Studio 2010 Ultimate Setup – Finish Page** to complete the installation successfully, as shown in Fig.C#-1.30:

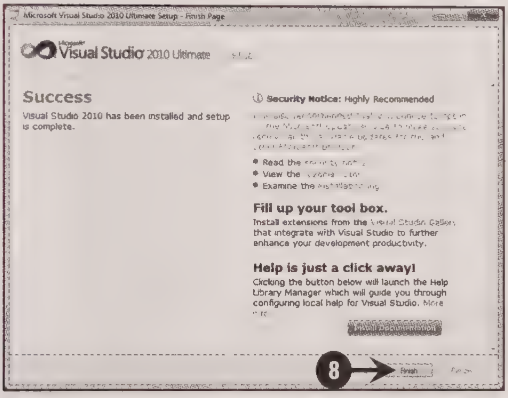


Fig.C#-1.30

After the successful installation, the Microsoft Visual Studio 2010 Setup dialog box appears (Fig.C#-1.31).

9 *Click* the **Exit** button on the **Microsoft Visual Studio 2010 Setup** dialog box to complete the setup, as shown in Fig.C#-1.31:

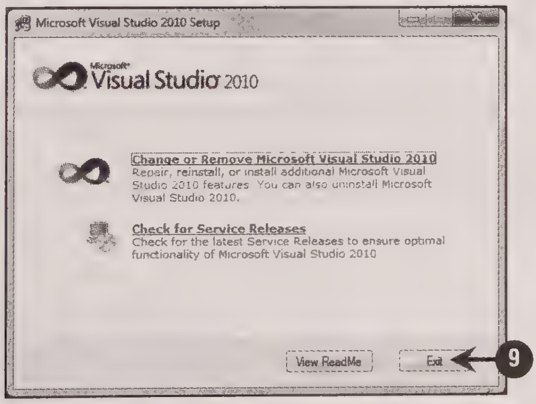


Fig.C#-1.31

Visual Studio 2010 is now successfully installed on your computer.

After installing Visual Studio 2010, let's explore the Visual Studio 2010 IDE.

Introducing the Visual Studio 2010 IDE

The Visual Studio 2010 IDE is a comprehensive environment to develop and execute.NET applications. It contains a menu bar, toolbars, and several windows that help you to design the UI of .NET applications as well as execute the applications.

Let's now learn to open the Visual Studio 2010 IDE.

Opening the Visual Studio 2010 IDE

After you have installed Visual Studio 2010 on your computer, you can open the application.

Perform the following steps to open the Visual Studio 2010 IDE:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010**, as shown in Fig.C#-1.32:

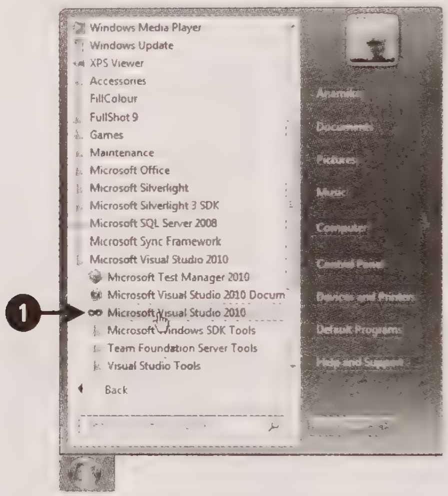


Fig.C#-1.32

The **Choose Default Environment Settings** dialog box appears (Fig.C#-1.33), prompting you to configure the IDE.

Note that the Choose Default Environment Settings dialog box is displayed only when you open Visual Studio 2010 for the first time. This dialog box allows you to set the default settings for Visual Studio 2010 by selecting the appropriate options. You can specify the default environment settings of Visual Studio 2010 to be either generic or specific to a .NET language, such as Visual C# or Visual Basic. For instance, if you select the General Development Settings option under the Choose your default environment settings list box in the Choose Default Environment Settings dialog box, the Visual Studio 2010 IDE is configured as a generic IDE and suitable for all the .NET languages. However, if you select the Visual C# Development Settings option, then the Visual Studio 2010 IDE is configured as an IDE specific to Visual C#.

2. *Select* an appropriate option from the Choose your default environment settings list box. In our case, we have selected the General Development Settings option (Fig.C#-1.33).
3. *Click* the **Start Visual Studio** button, as shown in Fig.C#-1.33:

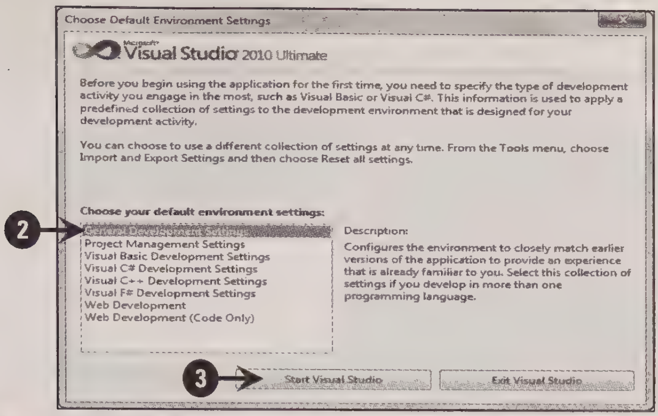


Fig.C#-1.33

The **Microsoft Visual Studio** message box appears, informing the user that the default environment for Visual Studio 2010 is being configured, as shown in Fig.C#-1.34:

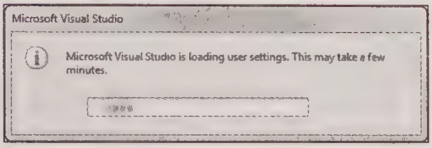


Fig.C#-1.34

After the default environment of Visual Studio 2010 is configured, the UI of Visual Studio 2010 appears, as shown in Fig.C#-1.35:

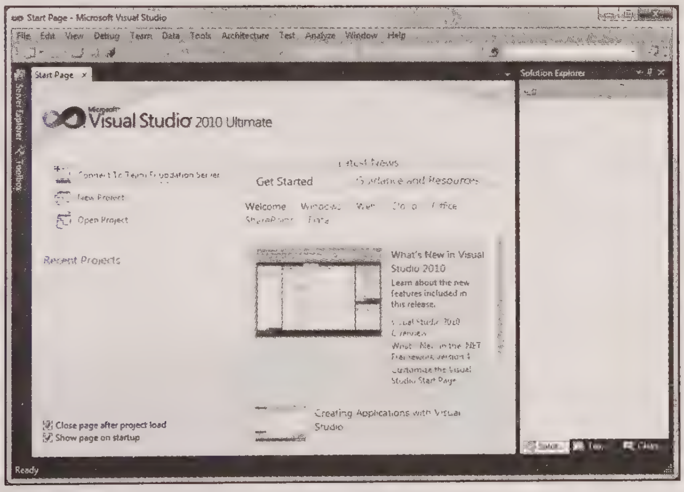


Fig.C#-1.35

Let's now explore the various components of the Visual Studio 2010 IDE.

Exploring the Visual Studio 2010 IDE

As stated earlier, the Visual Studio 2010 IDE enables you to develop various kinds of .NET applications, such as console, Windows Forms, and Web applications. The layout of the Visual Studio 2010 IDE is almost the same for all kinds of .NET applications, with only slight variations depending on the application.

Note

You learn how to create console, Windows Forms, and Web applications later in this chapter.

In this section, you learn about some of the essential elements of the Visual Studio 2010 IDE that are common for all kinds of .NET applications. In our case, we take the example of a Windows Forms application to learn about the IDE elements. Fig.C#-1.36 shows the Visual Studio 2010 IDE of a Windows Forms application:

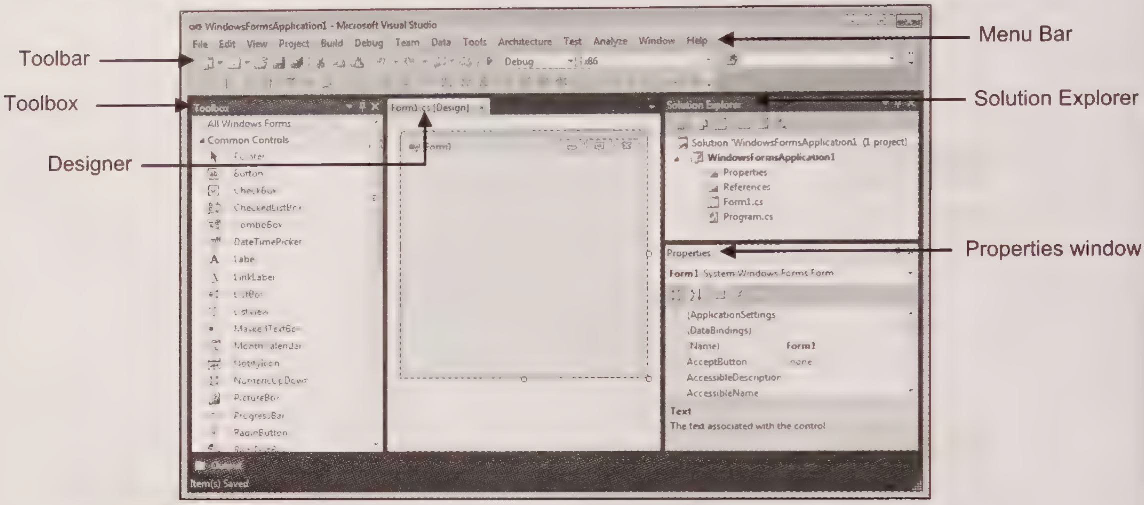


Fig.C#-1.36

Some of the important components of the Visual Studio 2010 IDE are as follows:

- The Start Page
- Menu Bar and Toolbars
- Solution Explorer
- The Properties window
- Toolbox
- Designer and Code Editor
- Server Explorer
- The Output window
- The Object Browser window
- The Class View window

Let's discuss these components one by one in the following sections.

Start Page

As the name suggests, Start Page (Fig.C#-1.35) is the first page that appears whenever you open Visual Studio 2010. This page contains various links to access the recently opened applications; the resources, articles, and news related to various products and technologies of Microsoft; and samples to create .NET applications and walkthroughs that help you to get started with Visual Studio 2010.

Menu Bar and Toolbars

The menu bar is a collection of menus, each of which contains a logical set of options to perform a particular task. The menus that are available in the Visual Studio 2010 IDE may vary in different kinds of .NET applications. However, some of the menus are common to all kinds of .NET applications, for example, the **File**, **Edit**, **View**, **Build**, **Debug**, and **Help** menus. Every menu provides options to perform a particular type of task.

For instance, the **File** menu contains options that are used to perform file management tasks, such as creating, opening, saving, and closing files (or applications). Fig.C#-1.37 shows the different menus present on the menu bar of a Windows Forms application:

Fig.C#-1.37

The menu bar of the Visual Studio 2010 IDE has a set of comprehensive options, some of which are more frequently used than others. To make it easier and quicker to access these menu options, the options are contained in toolbars. The toolbars appear just below the menu bar and have buttons as shortcuts for the most common menu options and other tasks. You can click the buttons on a toolbar to select an option or perform a particular task. Some of the common toolbars are **Standard**, **Build**, **Debug**, **Layout**, and **Help.** Fig.C#-1.38 shows the **Standard** toolbar, which appears by default in the Visual Studio 2010 IDE:

Fig.C#-1.38

Let's now learn about **Solution Explorer**.

Solution Explorer

In Visual Studio 2010, a .NET application is contained in a solution, which contains one or more projects, files, and other resources related to the application. Solution Explorer refers to a window in which the projects and files of a solution are displayed in the Visual Studio 2010 IDE. The window of Solution Explorer displays the projects and files in a hierarchical manner. Fig.C#-1.39 shows the Solution Explorer window for a Windows Forms application:

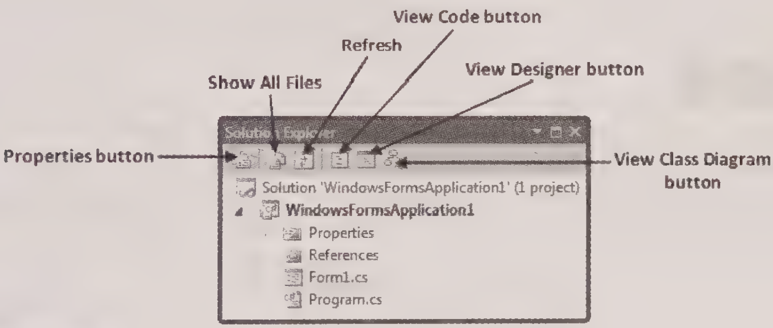


Fig.C#-1.39

As you can see in Fig.C#-1.39, the Solution Explorer window contains a toolbar with various buttons, such as Properties, Show All Files, Refresh, View Code, View Designer, and View Class Diagram, which perform specific actions. Among these, Show All Files, View Code, and View Designer are the most frequently used buttons. The Show All Files button displays all the files that are part of a solution. The solution in our case is WindowsFormsApplication1 (Fig.C#-1.39). The View Code button allows you to switch from the designer of the WindowsFormsApplication1Windows Forms application to its Code Editor. The View Designer button on the other hand, allows you to switch from Code Editor to the designer of the Windows Forms application. You will learn about the designer and Code Editor later in this chapter.

In Fig.C#-1.39, note that below the toolbar, the entire application is displayed as a hierarchy of various nodes. The root node at the top of the hierarchy has the same name as the solution and also displays the number of projects that the solution contains. This node generally has one or more child nodes, which represent the projects contained in the solution. If the solution has multiple projects, then every project will have child nodes for its references, files, and other resources.

Tip

If Solution Explorer is not visible in an application, then you can view it by selecting View→Solution Explorer on the menu bar or by pressing the CTRL+ALT+L keys in combination on the keyboard.

Next, let's learn about the Properties window in Visual Studio 2010 IDE.

The Properties Window

There are several projects, files, and resources that you work with while developing a .NET application in Visual Studio 2010. These projects and files have various attributes or properties, which you can view and set in the **Properties** window. The **Properties** window displays the properties of a solution, projects, files, and UI elements of a .NET application. Fig.C#-1.40 shows the properties of the **Form1** form in the **Properties** window, which appears when you select the **Form1** node in Solution Explorer and click the **Properties** button in the toolbar of Solution Explorer:

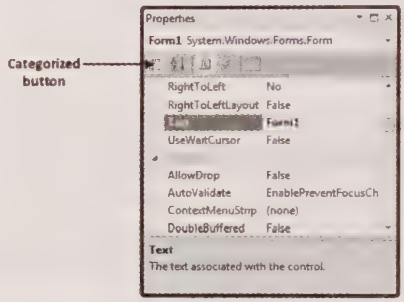


Fig.C#-1.40

In Fig.C#-1.40, you can see that the properties of the **Form1** form in the **Properties** window appear grouped in different categories, such as **Behavior**.

In Fig.C#-1.41, the properties of the **Form1** form are alphabetically grouped. You can group the properties of a form in alphabetical order by clicking the **Alphabetical** button on the toolbar of the **Properties** window, shown in Fig.C#-1.41:

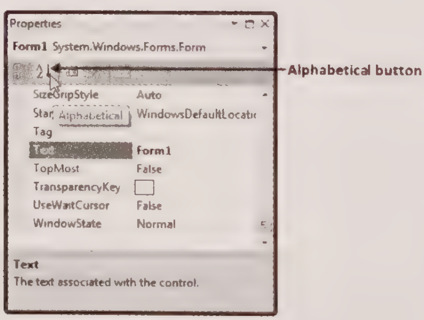


Fig.C#-1.41

Tip

You can open the Properties window by selecting View→Properties Window from the menu bar or by pressing the F4 key from the keyboard.

Next, let's learn about Toolbox in Visual Studio 2010 IDE.

Toolbox

Toolbox is a window that contains icons for various items and controls that you can add to a .NET application to design the UI of the application. The icons in Toolbox are logically grouped under different tabs, such as Common Controls, Menus & Toolbars, and Containers. Each tab contains the icons of the UI items and controls that pertain to a specific category; for example, the Containers tab contains the icons of those controls that allow you to group Windows controls, such as buttons, labels, text boxes, and radio buttons. Fig.C#-1.42 shows Toolbox containing the controls for a Windows Forms application:

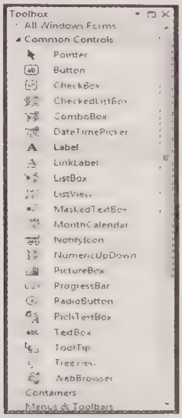


Fig.C#-1.42

To view the icons of the controls in a particular tab of Toolbox, you need to expand the tab by clicking it so that the triangle glyphs present on the left hand side of the name of the tab are expanded. The collapsed tabs, such as All Windows Forms and Containers (Fig.C#-1.42), have an empty triangle. Fig.C#-1.42 shows the Common Controls tab in its expanded form:

Tip

You can open Toolbox by selecting View→Toolbox from the menu bar or by pressing the CTRL+ALT+X keys in combination from the keyboard.

Now, let's learn about Designer and Code Editor in Visual Studio 2010.

Designer and Code Editor

As the name suggests, the designer helps you to design the UI of the application, while Code Editor helps you to add the code or functionality for the application. The available designer and Code Editor depend on the type of application and file with which you are working. Fig.C#-1.43 show the designer for a Windows Forms application: Visual Studio 2010 IDE offers windows called designer and Code Editor to help you develop .NET application easily and quickly.

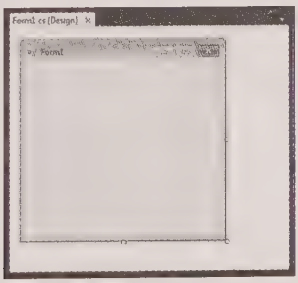


Fig.C#-1.43

Fig.C#-1.44 shows the Code Editor window for a Windows Forms application:

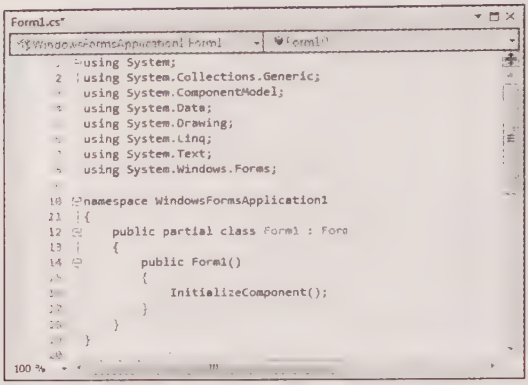


Fig.C#-1.44

Next, let's learn about Server Explorer component in Visual Studio 2010.

Server Explorer

Server Explorer refers to a window that you can use to connect to a database server and other services on a computer. In other words, Server Explorer allows you to create, modify, and close connections to databases. You can also add tables and query the databases or work with various system-specific services, such as event logging, message queues, and performance counters. Fig.C#-1.45 shows Server Explorer with connections to different databases and services on a local computer:

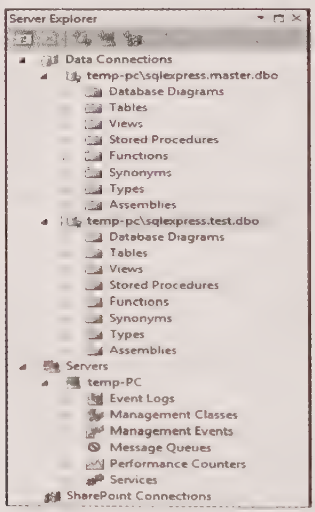


Fig.C#-1.45

Tip

To open Server Explorer, select View→Server Explorer from the menu bar or press the CTRL+ALT+S keys in combination from the keyboard.

Next, we discuss about the Output window in Visual Studio 2010.

The Output Window

When you compile or execute a .NET application, various status messages about the features in the Visual Studio IDE are generated and displayed in the Output window. This window displays these messages depending on the status of the compilation or execution of the application. Fig.C#-1.46 displays the Output window when a Windows Forms application is compiled or build:

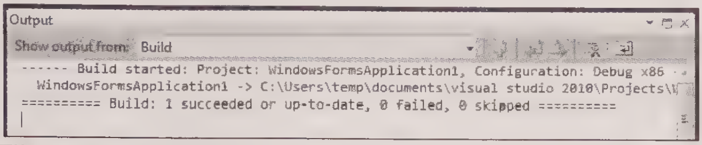


Fig.C#-1.46

You can specify whether or not you want to view the messages during compilation or execution by selecting the Build or Debug option from the drop-down list that appears on top of the toolbar of the Output window.

Tip

To open the Output window in an application, select View→Output from the menu bar or press the CTRL+ALT+O keys in combination from the keyboard.

Next, let's learn about the Object Browser window in Visual Studio 2010.

The Object Browser Window

The **Object Browser** window allows you to view the entities or objects, such as namespaces, classes, properties, methods, and events, which are available for an application. The Object Browser window has three panes: Object, Members, and Description. The Object pane appears on the left side, the Members pane appears on the upper-right side, and the Description pane on the lower-right side of the Object Browser window. Fig.C#-1.47 shows Object Browser window:

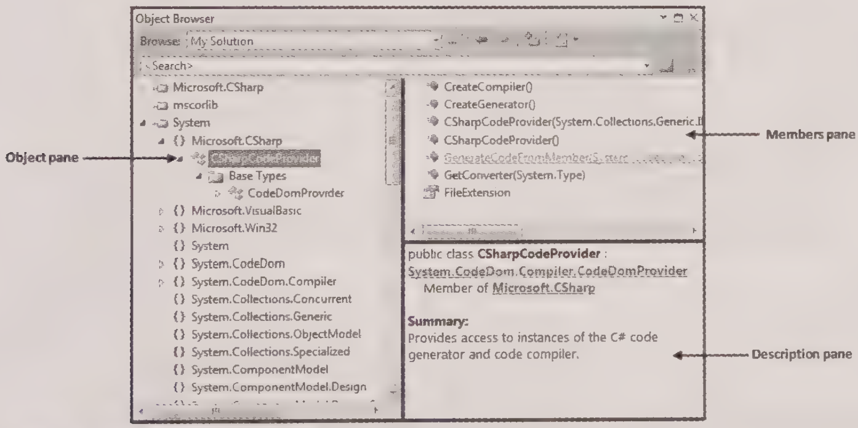


Fig.C#-1.47

In Fig.C#-1.47, the Object pane of the Object Browser window displays the assemblies, such as **System.Collections.Generic**, **System.Collections.ObjectModel**, and **System.ComponentModel** of the application in the form of nodes. You can expand the nodes by clicking them such that the empty triangle beside the selected node changes to a filled triangle. When you expand a node, you can view the namespaces that exist in the assembly in the form of child nodes of the expanded node. When you expand a namespace node, you can view the classes contained in the namespace. In addition, when you select a class, the properties and methods of the class appear in the **Members** pane, and a description of the class appears in the **Description** pane (Fig.C#-1.47).

Tip

To open Object Browser window, select View→Object Browser from the menu bar or press the CTRL+ALT+J keys in combination from the keyboard.

Next, let's discuss about the Class View window.

The Class View Window

The Class View window enables you to view items, such as assemblies, namespaces, classes, properties, and methods referenced or used in the current .NET application, in a hierarchical manner. The Class View window has two panes, Objects and Members, as shown in Fig.C#-1.48:

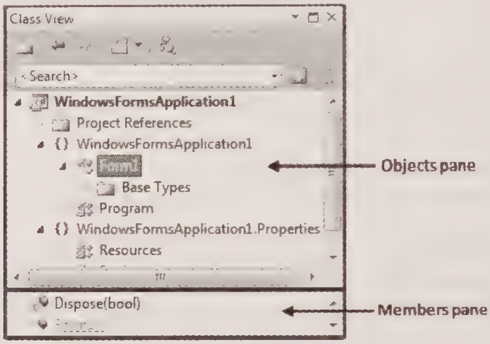


Fig.C#-1.48

As you can see in Fig.C#-1.48, the upper half of the Class View window is the Objects pane and the lower half is the Members pane. In the Objects pane, the projects of a solution form the top-level nodes of the hierarchical display. The child nodes of the project, WindowsFormsApplication1 represent the classes used in the project. Note that when you select a class node in the Objects pane of the Class View window, the properties and methods of the class are displayed in the Members pane.

Tip

You can open the Class View window by selecting View→Class View from the the menu bar orby pressing the CTRL+SHIFT+C keys in combination from the keyboard.

After getting familiar with the various components of the Visual Studio 2010 IDE, let's learn to create application using Visual Studio 2010. As stated earlier, Visual Studio 2010 offers a comprehensive and all-inclusive environment to allow you to develop various types of applications, such as console, Windows, and Web applications. Let's learn how to create these types of applications in the following sections.

Creating Simple Visual Studio 2010 Applications

Visual Studio 2010 provides several project templates to develop different types of .NET applications. The project templates offer an easy and convenient means of developing .NET applications because these project templates by default include the files and assemblies that are required in the application.

.NET applications can be broadly categorized as follows:

- Console applications
- Windows Forms applications
- Web applications

Let's learn to create, save and run these different types of applications, starting with a console application.

Creating, Saving, and Running a Console Application

Console applications refer to those .NET applications that do not have a GUI, apart from a command-line console. Users of a console application can interact with the application only through the command-line console, which means that data is displayed and read from the command-line. This makes console applications suitable in situations where user interaction is not required. Visual Studio 2010 offers an easy way to develop console applications by providing the appropriate project template.

Let's perform the following steps to create, save, and run a console application by using Visual Studio 2010:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar. **The New Project** dialog box appears (Fig.C#-1.49).
3. *Select* the **Visual C#** option in the **Installed Templates** pane. The installed project templates available for C# are displayed in the middle pane of the **New Project** dialog box (Fig.C#-1.49).
4. *Select* **Console Application** from the middle pane of the dialog box to create a console application (Fig.C#-1.49).
5. *Enter* an appropriate name for the console application in the **Name** text box. In our case, we have entered **MyFirstConsoleApplication** as the name (Fig.C#-1.49).
6. *Enter* the path of the folder for the console application in the **Location** combo box. In our case, we have entered the location as,**D:\C# 2010 Applications** (Fig.C#-1.49).
7. *Click* the **OK** button, as shown in Fig.C#-1.49:

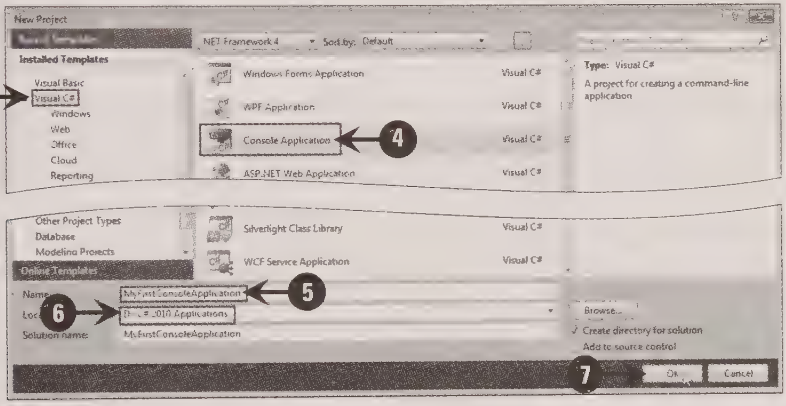


Fig.C#-1.49

The **MyFirstConsoleApplication** console application is created, as shown in Fig.C#-1.50:

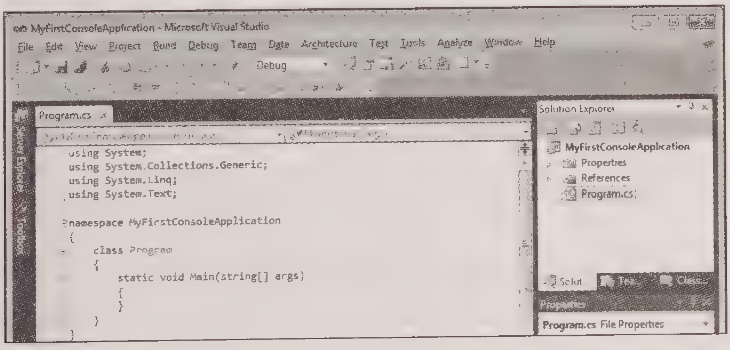


Fig.C#-1.50

8 *Add* the highlighted code shown in Listing 1.1, in the **Program.cs** file:

Listing 1.1: Adding Code in the Program.cs File of the **MyFirstConsoleApplication** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyFirstConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Welcome to C# Programming");
            Console.ReadLine();
        }
    }
}
```

In Listing 1.1, you can see that the **Main()** method has two lines of code. The first line—**Console.WriteLine("Welcome to C# Programming")**—accesses the **WriteLine()** method of the **Console** class, which exists in the System namespace in .NET Framework Base Class Library. When this line of code executes, the message, **Welcome to C# Programming**, is displayed on the command-line console. The second line of code—**Console.ReadLine()**—waits for the user to enter some text on the command-line console.

Note

The Console class allows you to read and display data from the command-line console.

9 *Select* **File→Save Program.cs** from the menu bar to save the file, as shown in Fig.C#-1.51:

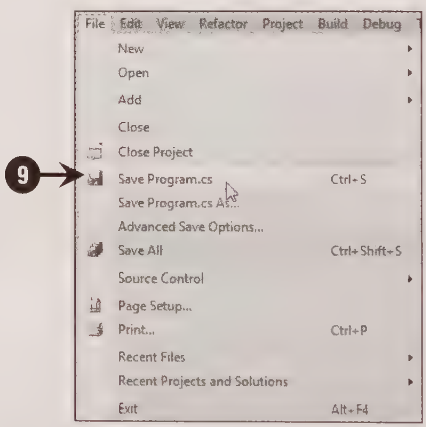


Fig.C#-1.51

Note

You can also save the changes in the Program.cs file (or the currently open file) by pressing the CTRL+S keys in combination from the keyboard. Similarly, you can press the CTRL+SHIFT+S keys to save the changes in all the files of an application.

To save the changes made in the entire application, select File→Save All from the menu bar.

10 *Select* **Debug→ Start Debugging** from the menu bar to run (or execute) the **MyFirstConsoleApplication**, console application, as shown in Fig.C#-1.52:

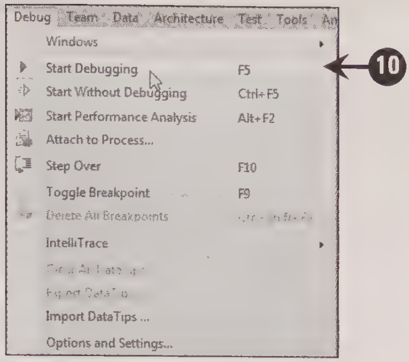


Fig.C#-1.52

The output appears, as shown in Fig.C#-1.53:

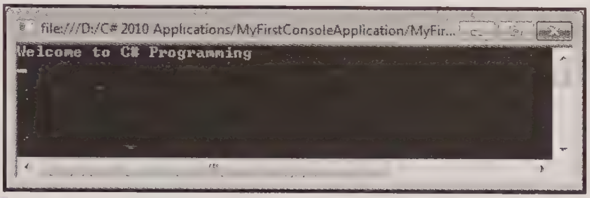


Fig.C#-1.53

As you can see in Fig.C#-1.53, the Welcome to C# Programming message is displayed on the command-line console.

Next, let's learn how to create, save, and run a Windows Forms application.

Creating, Saving, and Running a Windows Forms Application

Windows Forms applications refer to those applications that have a form at the center of the application. A form in a Windows Forms application is a rectangular area with a title bar at the top. In addition, the form has Minimize, Maximize, and Close buttons at its upper-right corner. A Windows form provides the GUI of the application, allowing interaction with the user.

Visual Studio 2010 provides a project template to develop a Windows Forms application.

Let's perform the following steps to create, save, and run a Windows Forms application by using Visual Studio 2010:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar, to display the **New Project** dialog box.
3. *Select* **Visual C#→Windows** in the **Installed Templates** pane on the left-hand side of the **New Project** dialog box, which displays the installed project templates available for Windows Forms applications in C#.
4. *Select* **Windows Forms Application** from the middle pane of the **New Project** dialog box to create a Windows Forms application.
5. *Enter* an appropriate name for the Windows Forms application in the **Name** text box. In our case, we have entered the name as **FirstWindowsFormsApplication**.
6. *Enter* the location to save the application in the **Location** combo box. In our case, we have selected the default location, which is **C:\users\temp\documents\visual studio 2010\Projects**.
7. *Click* the **OK** button.

The Windows Forms application named, **FirstWindowsFormsApplication,** is created, as shown in Fig.C#-1.54:

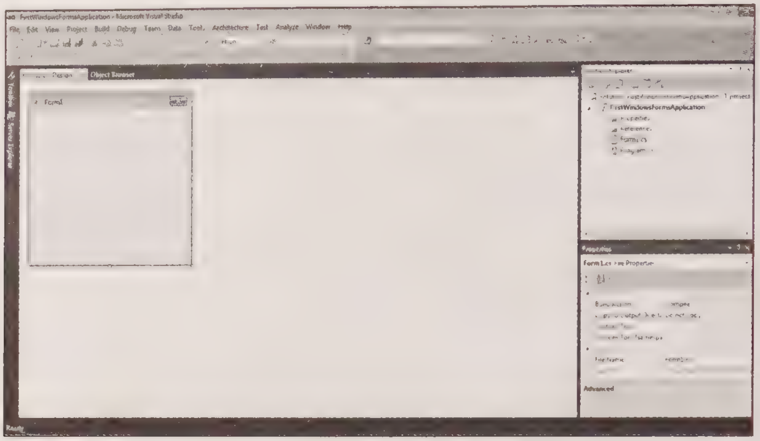


Fig.C#-1.54

As you can see in Fig.C#-1.54, the application has a rectangular area with Form1 displayed on upper-left corner and the Minimize, Maximize, and Close buttons displayed on its upper-right corner. This rectangular area is the form where you design the UI of the application.

Note

You learn more about Windows Forms applications in Chapter 5, Programming with Windows Forms Controls and Chapter 6, Working with Windows Forms, Menus, Toolbars and Dialog Controls.

8. *Double-click* the form in the Windows Forms designer, which opens the **Form1.cs** file in Code Editor, as shown in Fig.C#-1.55:

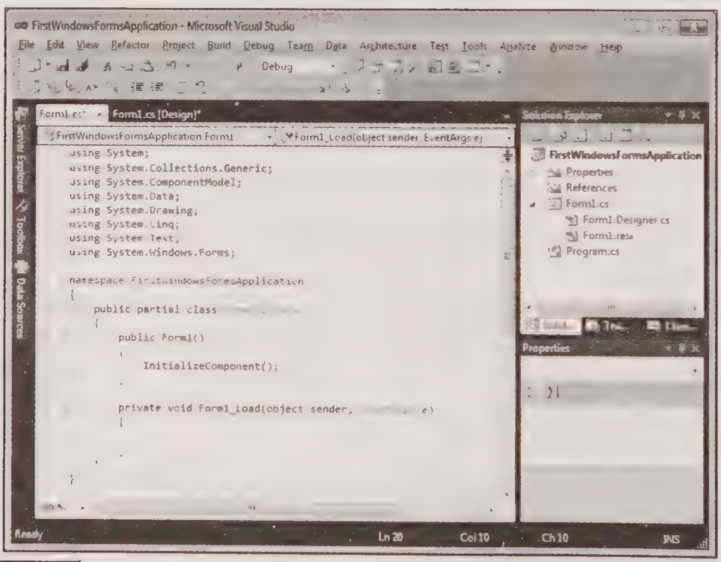


Fig.C#-1.55

9 *Add* the highlighted code shown in Listing 1.2 in the **Form1.cs** file, which contains the C# code for your Form1 form:

Listing 1.2: Adding Code in the **Form1.cs** File of the **FirstWindowsFormsApplication** Application

```
using System;using System.Collections.Generic;using System.ComponentModel;using
System.Data;using System.Drawing;using System.Linq;using System.Text;using
System.Windows.Forms;namespace FirstWindowsFormsApplicationprivate void
Form1_Load(object sender, EventArgs e)
{
    MessageBox.Show("Welcome to Windows Programming");
}
```

In Listing 1.2, the Form1 class represents the **Form1** form of the **FirstWindowsFormsApplication** application. Any functionality that you want to add to **Form1** needs to be added in the **Form1.cs** class. You can also see that the **Show()** method of the **MessageBox** class is accessed in the **Form1_Load()** method, which is the event handler for the **Load** event of **Form1**. The **Show()** method displays the **Welcome to Windows Programming** message when **Form1** loads at run time.

Note

You learn more about events and event handlers in Chapter 5, Programming with Windows Forms Controls.

10 *Select* **File→Save Form1.cs** from the menu bar to save the changes in the **Form1.cs** file, as shown in Fig.C#-1.56:

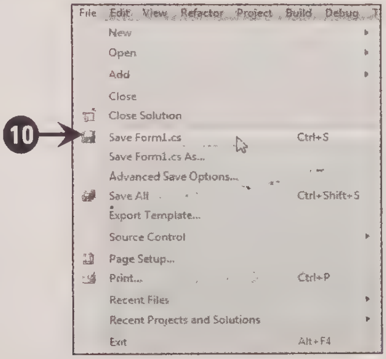


Fig.C#-1.56

11 *Select* **Debug→Start Debugging** from the menu bar to run the **FirstWindowsFormsApplication** application.

A message box with the message, **Welcome to Windows Programming** appears (Fig.C#-1.57).

Note

You can also press the F5 key from the keyboard to run an application.

12 *Click* the **OK** button on the message box, as shown in Fig.C#-1.57:

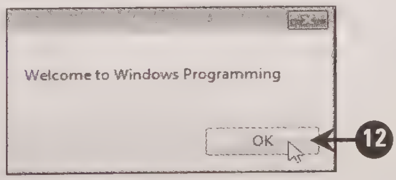


Fig.C#-1.57

The **Form1** Windows Form appears, as shown in Fig.C#-1.58:

Fig.C#-1.58

Next, let's learn how to create, save, and run a simple Web application by using Visual Studio 2010.

Creating, Saving, and Running a Web Application

ASP.NET is an essential part of the .NET Framework and includes all the services required to develop Web applications that involve minimal coding. The different types of ASP.NET applications that can be built in Visual Studio 2010 include Web sites, Web applications, Web services, and AJAX-enabled applications.

Let's learn to create a simple Web application by using Visual Studio 2010. Let's suppose that the Web application we create accepts some text as an input in a text box and displays it on a label after a user clicks a button.

Perform the following steps to create a Web application:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010**. Start Page of the Microsoft Visual Studio 2010 IDE opens.
2. *Click* the **New Project link** to create a new Web application.

   The New Project dialog box appears (Fig.C#-1.59).
3. *Select* **Visual C#→Web** in the **Installed Templates** pane of the **New Project** dialog box (Fig.C#-1.59).
4. *Select* **ASP.NET Empty Web Application** from the middle pane of the **New Project** dialog box (Fig.C#-1.59).
5. *Enter* a name for the Web application in the **Name** text box. In our case, we have named our Web application **MyFirstWebApplication** (Fig.C#-1.59).
6. *Enter* a location to save the Web application in the **Location** combo box. In our case, we have selected the default location, which is **c:\users\anamika\documents\visual studio 2010\Projects** (Fig.C#-1.59).
7. *Click* the **OK** button, as shown in Fig.C#-1.59:

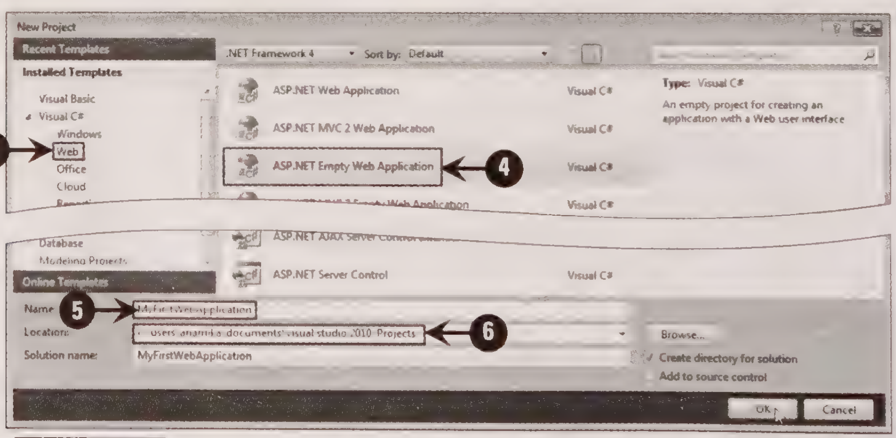


Fig.C#-1.59

The **MyFirstWebApplication** Web application is created, as shown in Fig.C#-1.60:

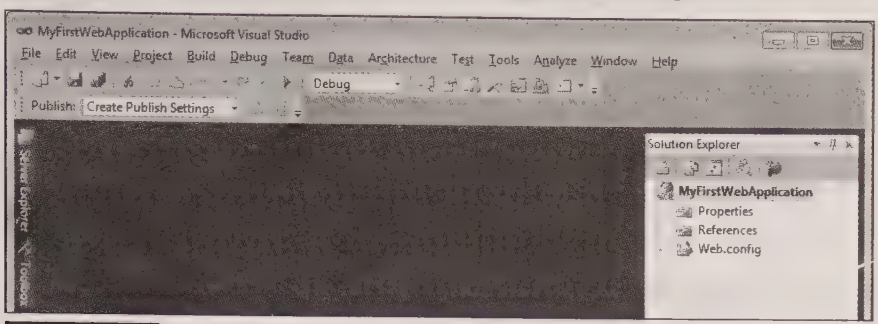


Fig.C#-1.60

Note that there are no Web forms in **Solution Explorer**. Therefore we must explicitly add one Web form to the newly created Web application. For this, proceed to the next step.

8 *Right-click* the Web application in **Solution Explorer** and *select* **Add→New Item** from the context menu, as shown in Fig.C#-1.61:

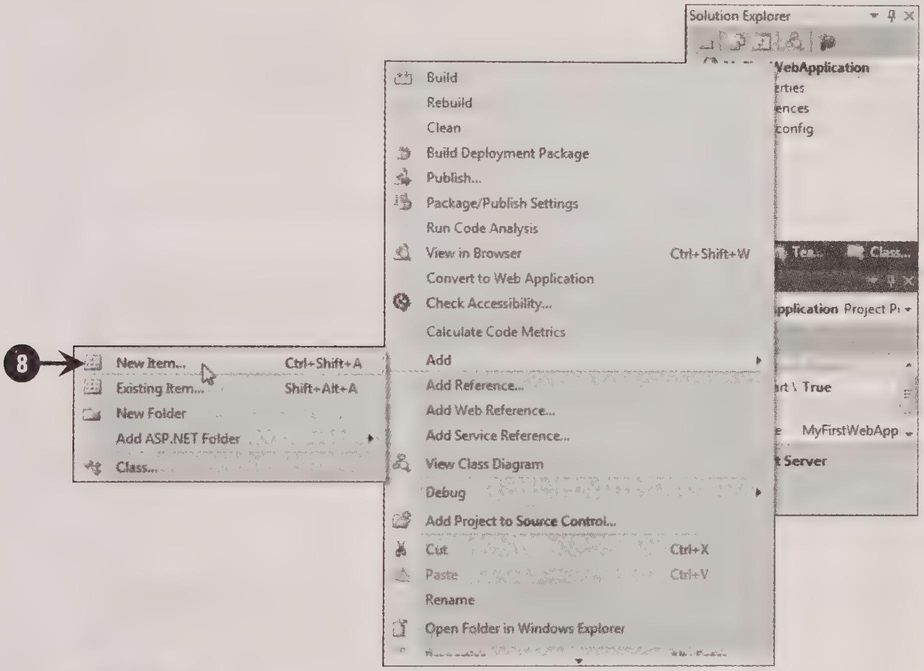


Fig.C#-1.61

The **Add New Item** dialog box appears.

9 *Select* the **Web** option from the **Installed Templates** pane in the left-hand side of the **Add New Item** dialog box (Fig.C#-1.62).

10 *Select* the **Web Form** option from the middle pane of the **Add New Item** dialog box (Fig.C#-1.62).

11 *Click* the **Add** button to add the Web form in the **MyFirstWebApplication**, application, as shown in Fig.C#-1.62:

Fig.C#-1.62

The **Source** view of the Web form appears (Fig.C#-1.63).

12 *Click* the **Design** tab located at the bottom of the **MyFirstWebApplication** window, as shown in Fig.C#-1.63:

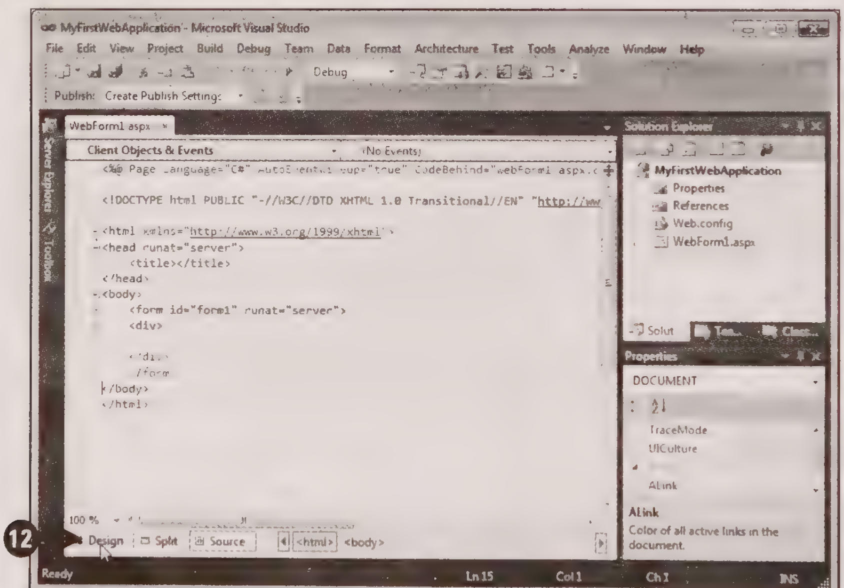


Fig.C#-1.63

The **Design** view of the Web form appears, as shown in Fig.C#-1.64:

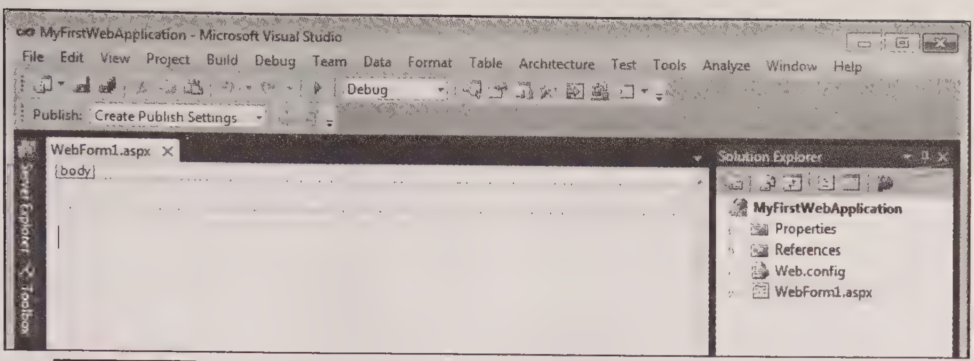


Fig.C#-1.64

13. *Open* **Toolbox** and *double-click* the **Label** control to add the control on the **WebForm1.aspx** Web form.
14. *Double-click* a **TextBox** control and a **Button** control to add on the Web form. The **WebForm1.aspx** Web form now appears with the three controls, as shown in Fig.C#-1.65:

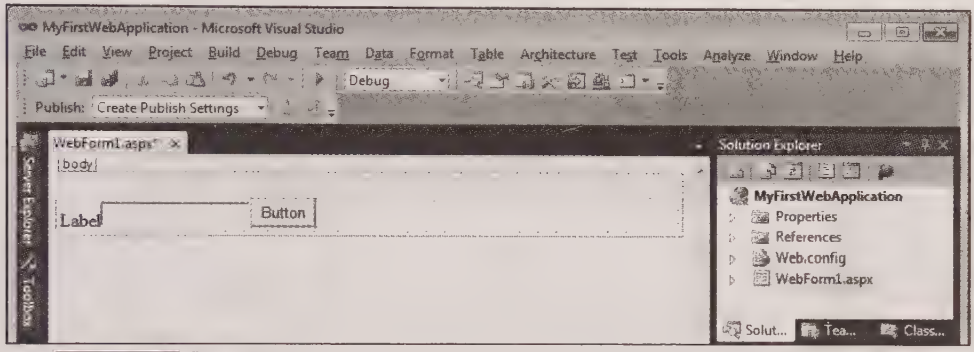


Fig.C#-1.65

Table 1.2 describes the controls that we added on the Web form, **WebForm1.aspx**:

Table 1.2: Controls Added to the WebForm1.aspx Web Form

Control Name	Description
textBox1	Accepts input from the user
label1	Displays the name entered in the TextBox control, textBox1, concatenated with a welcome text
button1	Assigns text from the TextBox control, textBox1, to the Label control, label1

When you add a Label control on the **WebForm1.aspx** Web form, the form displays **Label** as the default text for the control. Similarly, a Button control displays **Button** as the default text.

Note

You can change the default text of a control through the Text property of the control in the Properties window.

15. *Select* the **Label** control and press the **F4** key from the keyboard. The **Properties** window appears for the selected **Label** control (Fig.C#-1.66).
16. *Set* the **Text** property of the **Label** control to **HELLO WORLD FROM:**, as shown in Fig.C#-1.66:

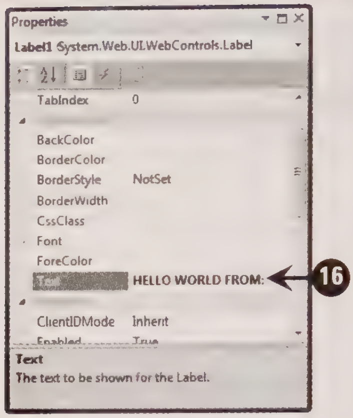


Fig.C#-1.66

17 *Set* the **Text** property for the **button1** control to **CLICK HERE**.

After setting the properties of the controls, the **WebForm1.aspx** Web form appears, as shown in Fig.C#-1.67:

Fig.C#-1.67

18 *Double-click* the **Button** control in the **Design** view. Code Editor for the Web form appears, as shown in Fig.C#-1.68:

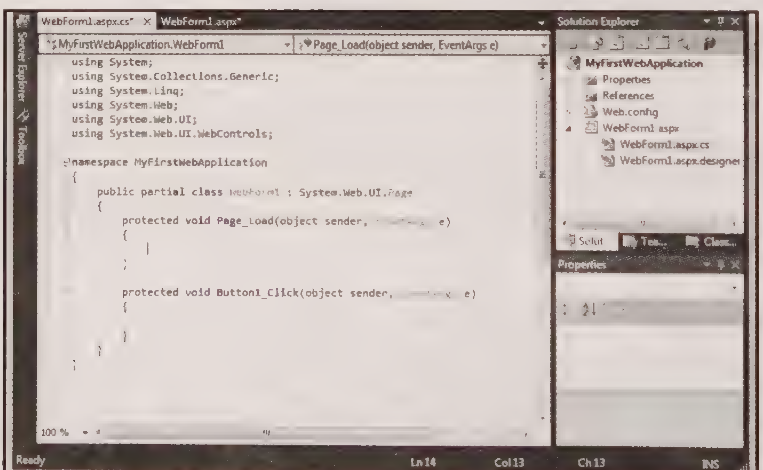


Fig.C#-1.68

19 *Add* the highlighted code shown in Listing 1.3, in the **WebForm1.aspx.cs** file:

Listing 1.3: Displaying the Code for the Click Event of Button1

```
protectedvoid Button1_Click(object sender, EventArgs e)
{
      Label1.Text += TextBox1.Text;
}
```

In Listing 1.3, the **Text** property of the **label1** control is assigned the text contained in the **Text** property of the **textBox1** control. As we want to retain the initial text that is displayed on the **label1** control, a new text specified in the **textBox1** control is appended to the **label1** control by using the addition assignment operator, +=.

20 *Execute* the application by *pressing* the **F5** key. The output appears, as shown in Fig.C#-1.69:

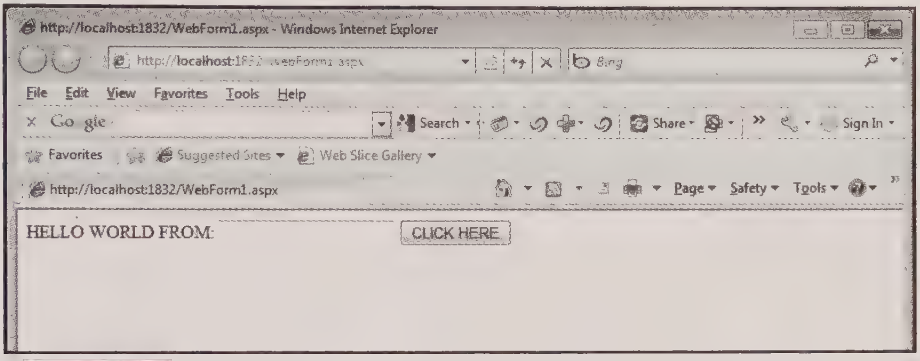


Fig.C#-1.69

As you can see in Fig.C#-1.69, when the **Web From1.aspx** Web form, is loaded in the browser, the **Label** control displays the text **HELLO WORLD FROM:** and the **Button1** control displays **CLICK HERE**.

21 *Enter* a name in the **textBox1** text box and *click* the **CLICK HERE** button (Fig.C#-1.70).

The name you entered in the **textBox1** text box is displayed along with the text ***HELLOWORLD FROM:***, as shown in Fig.C#-1.70:

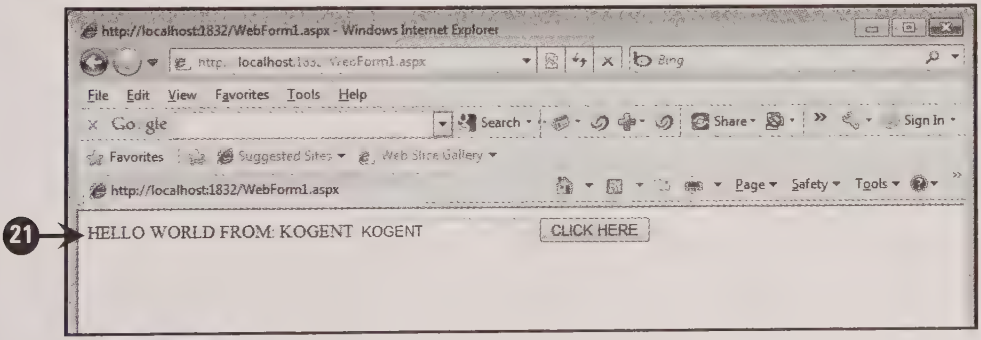


Fig.C#-1.70

You have successfully created an ASP.NET Web application.

With this, we come to the end of the chapter. Let's now summarize the main points of the chapter.

Summary

In this chapter, you have learned about:

- Advantages of .NET Framework 4.0
- Architecture of .NET Framework 4.0
- Components of .NET Framework 4.0 such as CLR, assemblies, and Windows Forms
- Main features and enhancements of Visual Studio 2010
- Installing Visual Studio 2010 on your computer
- Components of the Visual Studio 2010 IDE
- Creating different types of Visual Studio 2010 applications

Chapter 2

Introducing C# 2010 Programming Essentials

In this Chapter:

- Explaining the Relationship between C# and .NET Framework 4.0
- Describing C# 2010 Language Features
- Exploring C# 2010 Keywords
- Explaining Data Types
- Working with Variables and Constants
- Working with Operators and Operator Precedence
- Working with Strings
- Implementing Type Safety
- Creating Enumerations
- Working with Arrays

There is a plethora of programming languages available these days but a user prefers a language that involves less coding and at the same time enables them to develop applications in an interactive and visually powerful environment, such as Visual Studio 2010 Integrated Development Environment (IDE). C# is a language that involves less coding and has a rich Graphical User Interface (GUI).

C# is a high-level programming language, which is easy to learn and understand as compared to a low-level programming language. A low-level programming language, also known as machine language, is difficult to learn and interpret. In addition, it is designed for a specific computer to reflect its machine code. On the contrary, a high-level language is easy to read and close to spoken language. In C#, you can write source code using statements, which are similar to English words and statements; therefore, people choose high-level programming languages, such as C#, Java, and C++ for programming purposes.

In this chapter, you learn how C# is related to .NET Framework 4.0, its new features, such as dynamic types, named and optional arguments, covariance, and contravariance of C# 2010. Next, the chapter discusses C# 2010 keywords, data types, variables and constants, operators and their precedence, string operations, type conversions, boxing and unboxing, enumerations, and arrays.

Now, let's discuss about C# and .NET Framework 4.0 first.

Explaining the Relationship between C# and .NET Framework 4.0

C# is an object-oriented programming language developed by Microsoft. It helps application developers to build secure and robust applications that target the .NET Framework. It is simple and easy to learn and implement C#, as it has syntax and control structures similar to C, C++, and Java. Moreover, being an object-oriented programming language, it fully supports the concepts of encapsulation, inheritance, and polymorphism. In addition to its object-oriented features, C# has various other features, such as Language-Integrated Query (LINQ), delegates, and lambda expressions.

The .NET Framework on the other hand, forms an essential component of Windows that includes a vast set of standard class libraries, which can be used by many programming languages, such as C#, Visual Basic .NET, and J#. It also includes a runtime environment known as the Common Language Runtime (CLR), which provides a runtime environment for the execution of applications that have been written using any of the .NET programming languages.

Though C# is not a part of the .NET Framework, yet all the source code written in C# always runs within the .NET Framework. C# has been created to be used with .NET Framework; therefore, Microsoft has always attempted to co-evolve both, the C# language and .NET Framework.

Let's discuss the new features of C# 2010.

Describing C# 2010 Language Features

C# 2010 has improved to a great extent from its predecessors, as it now includes features such as dynamic programming, which is the major theme of C# 2010. New compiler features, covariance and contravariance, and improvements in Component Object Model (COM) interoperability are the other enhancements in C# 2010. These features are added to increase the productivity of C# developers and help them create C# applications more easily and efficiently. Following is a list of new features that are added to C# 2010:

- Dynamic types
- Named and optional arguments
- Enhanced COM Interoperability
- Covariance and Contravariance
- New command-line compiler options
- Implicit line continuations

Now, let's discuss all these features in detail.

Dynamic Types

C# 2010 aims at providing the application developers more flexibility while developing applications in C#. It has introduced the new **dynamic** type on which, you can perform any operation, such as string manipulation or arithmetic operation. The dynamic types are resolved at runtime; therefore, you do not need to worry about the type of a variable while assigning values that are returned from methods or on evaluation of an expression. The CLR automatically performs necessary binding for you according to the data type of returned value.

Let's consider the following code snippet:

```
dynamic d = 5;
var total = d + 2;
Console.WriteLine(total);
```

In the preceding code snippet, a **dynamic** type variable **d** is declared, which has an integer value, 5. In addition, the **var** type variable **total** is used to store the sum of the value of **d** and 2. While declaring a variable using the **var** type, you do not need to define a data type for it. The CLR decides the data type for a variable of type **var** on the basis of the value assigned to it. The data type for a variable of type var is resolved at compile time whereas the data type for a variable of type dynamic is resolved at runtime.

Using the **dynamic** type, programmers can now write method and indexer calls, property and field accesses, and even object invocations, which can bypass the C# type checking process and get inferred during runtime. This feature is beneficial while working with various kinds of objects such as COM.

Note

The result of all the dynamic operations is also dynamic.

Next, let's learn about the support for named and optional parameters introduced in C# 2010.

Named and Optional Arguments

C# 2010 has introduced support for named and optional arguments. These are two distinct features that have been added to C# 2010. Named arguments allow you specify an argument for a specific parameter by associating the argument with the name of the parameter instead of passing the position of the parameter in the parameter list. On the other hand, in the optional arguments, you can omit arguments for the parameters that already have some defined default value. You can use both named and optional arguments with methods, indexers, constructors, and delegates. The evaluation of arguments takes place in the order in which they appear in the argument list and not in the order they appear in the parameter list while using both named and optional arguments.

For instance, let's consider an example where you need to perform a basic arithmetic operation, such as multiplication. In this case, you need to pass three parameters to a method called **Product()**, as shown in the following code snippet:

```
static int Product (int a, int b, int c = 6)
{
    return a * b * c;
}
```

The preceding code snippet shows that **c** is an optional parameter since a default value is specified to it inside the **Product()** method.

Note

An optional parameter must always be declared as the last parameter in the parameter list after declaring the required parameters.

You can call the **Product()** method without specifying the optional argument, as shown in the following code snippet:

```
Product (5 , 7);
```

Now, let's consider an application where you need to perform subtraction between two numbers, as shown in the following code snippet:

```
static int Subtract(int a, int b)
{
      return a - b;
}
```

The method created in the preceding code snippet can be called, as shown in the following code snippet:

```
Subtract(b: 3, a: 5);
```

The preceding code snippet illustrates that named arguments can be passed to parameters in the order **a,b** or **b,a**.

Enhanced COM Interoperability

COM Interoperability is a technology that has been included into the .NET Framework through which the COM objects can easily interact with .NET objects, and vice versa. Whenever a COM component is registered, the .NET Framework automatically creates a type library and special registry entries. The previous versions of .NET Framework require a large number of explicit type casts and excessive usage of **ref** keyword. However, with .NET Framework 4.0, you can easily interoperate with COM objects from the C# source code. The fewer typecasts required because of introduction of **dynamic** types, support for named and optional arguments, and less dependency for **Primary Interop Assemblies (PIAs)** are attributed to the enhanced COM interoperability in C# 2010. In addition, you can avoid using the **ref** keyword in case you need to pass an argument by reference.

Note

A PIA is a collection of the type definition of all the types that have been implemented with COM objects. A COM type can be described by any number of interop assemblies; however, only one interop assembly is tagged as PIA. The publisher of the original COM type signs PIA and includes some customizations that are intended to make types easier to use from managed code.

Next, let's learn about the support for covariance and contravariance provided in C#.

Covariance and Contravariance

The support for covariance and contravariance in C# 2010 facilitates implicit type safe conversions for array types, delegate types, and generic type arguments. Using covariance, you can always assign a smaller type to a larger type, such as assigning a **decimal** type to an **object** type. While with contravariance, the assignment compatibility that is preserved in covariance is totally reversed. For instance, you can treat an **object** type as a **decimal** type by assigning covariance to a **decimal** type. You can use the **out** keyword to implement covariance, while **in** keyword is used for implementing contravariance.

Note

The concept of generic type was introduced in C# 2.0 so that the binding of a type (declared as a generic type) to a data type can be deferred until generic type is not used. Classes, methods, events, delegates, and interfaces can be declared and defined as generic. Generic type helps to enhance performance, promote code reusability, and type-safety.

New Command-Line Compiler Options

You can always compile and run C# applications from command-line. The C# compiler typically generates executable (.exe) files and dynamic link libraries (.dll) files. Besides the various command-line compiler options available in C#, the new version introduces a new command-line compiler option called **/langversion**. If you use the **/langversion** option, the compiler allows you to execute the syntax that is valid for a specified version of C#. In addition, by using the **/appconfig** compiler option, a C# application can provide the location of application configuration file of an assembly to the compiler.

Tip

You can get more help on command-line options by typing csc -help on the command-line.

Next, let's learn about implicit line continuations.

Implicit Line Continuations

There may be situations where you might need to continue a single statement over multiple lines. In such situations, you can implicitly continue a statement in the next line, as shown in the following code snippet:

```
Console.WriteLine(
CalculateArea(length: 10, breadth: 5));
```

Now, let's proceed towards exploring keywords in C# 2010.

Exploring C# 2010 Keywords

Keywords are those words or identifiers, which are reserved to be used for a specific task. You cannot use a keyword to define the name of a variable or method. Keywords are the reserved words whose meanings are predefined to the C# compiler. They are used in all the programming languages. A keyword is an essential part of a language definition. However, there are some keywords that have special meaning in the context of code but such keywords are not reserved keywords. These keywords are known as contextual keywords. Table 2.1 lists the reserved and contextual keywords:

Table 2.1: Reserved and Contextual Keywords in C# 2010

Reserved Keywords			
abstract	as	base	bool
break	byte	case	catch
char	checked	class	const
continue	decimal	default	delegate
do	double	else	enum
event	explicit	extern	false
finally	fixed	float	for
foreach	goto	if	implicit
in	in (generic modifier)	int	interface
internal	is	lock	long
namespace	new	null	object
operator	out	out (generic modifier)	override
params	private	protected	public
readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc
static	string	struct	switch
this	throw	true	try
typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual
volatile	void	while	
Contextual Keywords			
add	alias	ascending	descending
dynamic	from	get	global

Table 2.1: Reserved and Contextual Keywords in C# 2010			
group	into	join	let
orderby	partial (type)	partial (method)	remove
select	set	value	var
where (generic type constraint)	where (query clause)	yield	

In Table 2.1, the reserved and contextual keywords are given. These keywords can be used in C# applications to develop a scalable application.

Note

You can use keywords as identifiers in your program if they include @ as a prefix. For instance, @try is a valid identifier but you cannot use try as an identifier because it is a keyword.

Now, let's learn about the data types in C# 2010 in the next section.

Explaining Data Types

C# is a strongly typed and rich language; therefore, every variable in C# is associated with a data type. Data types specify the size and type of values that can be stored. There are a number of data types available in C# that allow a programmer to select the data type appropriate to the requirement of the application. The data types in C# are primarily categorized in the following three parts:

- Value types
- Reference types
- Pointer types

Fig.C#-2.1 shows the categorization of C# data types:

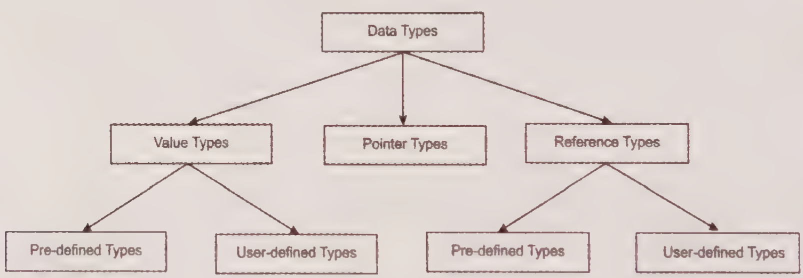


Fig.C#-2.1

Let's first start discussing value types.

Value Types

Value types have fixed length and are stored on the stack of memory. A stack is used to store certain information in a computer. It is basically a data structure, which follows the last-in-first-out principle. A stack is used to store value type variables in C#. When a value of a variable is assigned to another variable, the value is copied from one variable to another. This means that two identical copies of the value are available in the memory.

Table 2.2 lists all the value types in C#:

Table 2.2: Value Types in C#

Value Type	What a Value Type Represents	Ranges	Default Value
bool	Boolean type, represents Boolean value	True or False	false
byte	8-bit unsigned integer type	0 to 255	0
char	Single 16-bit Character type, a char value is a Unicode character	U+0000 to U+ffff	'\0'
decimal	128-bit Precise decimal type with 28-29 significant digits	$\pm 1.0 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$	0.0M
double	64-bit Double-precision floating-point type	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	0.0D
float	32-bit Single-precision floating-point type	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	0.0F
int	32-bit signed Integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integral type	-32,768 to 32, 767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

Value types can further be categorized into two parts: Struct types and Enumeration types, which are discussed next.

Struct Types

The struct type is a value type and is stored on a stack. These types can encapsulate a small group of related variables, such as the name, rollno, age, class, and section of a student. The members of struct types, such as constructors, methods, properties, operators, events, nested types, indexers, constants, and fields cannot be declared as **protected**.

Moreover, struct types are created using the **struct** keyword. When you create a **struct** object and assign it to a variable, the variable holds the complete value of the **struct** object. As you know struct type is a value type; so any changes made in the new copy of struct type variable do not affect the old copy.

The struct types are categorized into the following five categories:

- **Integral type:** Implies that C# uses a set of basic data types, which are known as integral types. This set consists of the types, such as **sbyte**, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**, and **char**. Integral types hold whole numbers, such as 123, -123, and 3456.
- **Floating-Point type:** Represents values in fractions, for example -123.45 and 123.56. The floating-point types are divided into two types: **float** and **double**.

Note

To specify a number to a float type, you must append the character F (or f) to a value, such as 12.4F. If you omit F, the value is treated as double. The double data type contains a double-precision floating point number. It stores 64 bit floating-point values.

- **Decimal type:** Specifies that **decimal** types are equivalent to long integer types, which behave as a floating-point type. Decimal values are intended to be used for the calculations in which truncation beyond the

decimal point is not desired. The decimal types have more precision and smaller range as compared to the floating-point types. You must append character M (or m) to a value such as 123.45M to specify a number as a decimal type. If you omit M, the value is treated as double.

- **Boolean types (bool):** Allows you to test a particular condition during the execution of a program. Boolean type takes only two values, **True** or **False**.
- **Nullable types:** Allows a user to assign a null value to a variable of any data type. The nullable types are often used with databases.

Enumeration Types

An enumeration provides a way for attaching the integer type numbers to the names in a program, which in turn increases the readability of a code. Enumerations are the user-defined integer data types that are declared using the **enum** keyword. An enumeration makes code easier to maintain by assigning legitimate values to it. Moreover, enumeration is the most basic user-defined type, which makes a list of possible values for a given type. For instance, you might have a program that models a traffic light. Traffic lights have three possible states, the green (or go) light, the yellow (or warning) light, and the red (or stop) light; therefore, at any given point of time, a traffic light can be red, yellow, or green.

Now, let's learn about reference types in C#.

Reference Types

Reference types have varied lengths and are stored on the heap of memory. When a value is assigned to a reference type variable, only the reference is copied and the actual value remains the same in the memory location. We can further categorize the reference types into two parts, pre-defined and user-defined types.

Note

Heap is a data structure that is used to store reference types, such as objects in C#. Objects are stored arbitrarily in the heap and their existence or size cannot be determined until the application starts running.

Let's discuss pre-defined and user-defined types next.

Pre-defined Types

Pre-defined reference types can be divided into two types:

- **Object type:** Refers to the ultimate base type of all the other pre-defined and user-defined data types in C#. You can use the **Object** type to convert a value type on a stack into an **Object** type to be placed on the heap.
- **String type:** Refers to the creation and manipulation of strings in C#. You can perform various operations, such as copy, compare, and concatenation on strings.

User-defined Types

User-defined reference types refer to those types, which are defined using the pre-defined types. Following are the user-defined reference types:

- **Class:** Specifies that C# is purely an object-oriented language, so the base of all the C# programs is class. Using classes, you can group logically related data items and functions, which are required to work on data items.
- **Interface:** Refers to the collection of members, such as methods, delegates, events, and properties, which are implemented by classes. Interfaces cannot be instantiated but you can use them to offer a set of functionalities that is common to several other classes. The concept of interface was introduced to support multiple inheritance because in C#, a class does not support multiple inheritance.

Note

Extending the functionalities and behavior of a class in another class is called Inheritance.

- **Delegate:** Refers to the objects that store references of the methods whose signatures match with the signature of the delegate. Delegates are the reference types that encapsulate a method with a specific signature.
- **Array:** Refers to an ordered arrangement of data elements. It is a data structure that contains a number of variables of the same data type at contiguous locations of the memory.

Now, let's learn about pointer types in C# 2010.

Pointer Types

To maintain type safety and security in the context of references, C# uses delegates and does not support pointers. However, you can use the **unsafe** keyword to define an unsafe context where you can use pointers. In other words, pointer types are used only in the unsafe context. Pointer types are not inherited from the **Object** base class, which is the base class of all .NET types. You cannot convert pointer types to **Object** types and vice-versa. A pointer variable is used to store the address of another variable. The following syntax displays the declaration of pointer type:

```
datatype* identifier;
```

You have learned about data types in C#, further you learn about variables and constants.

Working with Variables and Constants

A variable is an identifier that denotes a storage location in memory. Using the name of a variable in a program you can refer to the information stored at a particular location. Unlike, the constants that remain unchanged during the execution of a program, a value of a variable can be changed any time. Every variable has a type that determines the values to be stored in it.

You can give any name to a variable but it should be meaningful because it makes the code more readable. Some examples of meaningful variable names are salary, height, name, age, and total_marks.

Following are certain rules for naming C# variables:

- The name of the variable may consist of letters, digits, and the underscore character (_).
- The name of the variable must not begin with a digit.
- The variable names using the uppercase and lowercase characters are distinct; for example, salary and SALARY are two different variables.
- The variable name should not be a keyword.
- The variable name should not contain spaces.

Table 2.3 shows the examples of valid and invalid variable names:

Table 2.3: Some Examples of Valid and Invalid C# Variable Names

Variable Name	Status
salary	Legal
Total_amount	Legal
yearly_cost	Legal
_2010_tax	Legal, but not advised
Total#amount	Illegal; contains the illegal character #
double	Illegal; double is a keyword
9number	Illegal; first character is a digit

On the other hand, constants are those values which are fixed and do not change during the execution of a program. For instance, the value of Pi is fixed, so it can be declared as a constant. Moreover, any value that you do not want to change can be declared as a constant. Let's learn to declare a variable in C#.

Declaring Variables

Any information in a program that can vary must be stored in a named memory location called variable. After specifying a meaningful name and appropriate data type to a variable, it is declared in the program so that the compiler can easily recognize the variable declared with a specific data type.

Note

A compiler is software that converts a source code written in a programming language into a machine language, which a computer can understand.

A variable declaration performs the following tasks:

- Tells the compiler the name of the variable
- Tells about the type of data a variable holds
- Decides the scope of a variable

Moreover, a variable must be declared before it is used in a program. To declare a variable, you must specify a data type to it according to the value assigned, followed by the name of the variable. The following is the syntax for declaring a variable:

```
datatype variable1, variable2, variable3,… variableN;
```

If the variables to be declared are more than one and have same data types, then you can declare the variables in the same line separated by a comma, as shown in the preceding syntax. A code line in C# always ends with a semi colon. The following are some examples of legal variable declaration:

- float length;
- int age;
- char b1, b2;
- double pi;
- decimal money;

Assigning Values to Variables

After the declaration of a variable, you need to initialize its value. Initialization of a value to a variable means assigning a value to a variable. While initializing a variable, ensure that a variable is assigned a value before being used in a program. It is not mandatory to initialize a variable at the time of declaration. The syntax for initialization is as follows:

```
Variablename = value;
```

or

```
variable1 = variable2 = variable3 = value;
```

Following is the syntax to initialize a variable at the time of declaration:

```
datatype variablename = value;
```

or

```
datatype variable1 = value, variable2 = value;
```

The following are some valid examples of initializing a variable:

```
int age = 23;
bool status = false;
float height = 164.5f;
char sex = 'F';
int a = 4,b = 6;
```

```
decimal d = 2.34M;
```

Let's learn to declare constants.

Declaring Constants

As discussed earlier, constants are fixed values that cannot be changed during the execution of a program. Following is the syntax to declare a constant:

```
const datatype constantname = value;
```

You must declare and then initialize a constant, as shown in the following code snippet to avoid an error:

```
const float PI = 3.14;
const int count = 4;
```

The values assigned to constants cannot be changed during the execution of a program. A constant can also be initialized using an expression, as shown in the following code snippet:

```
const int length = 10;
const int breadth = length * 2;
```

Moreover, you cannot use a non-constant value in an expression, which is used by a constant, as shown in the following code snippet:

```
int length = 10;
const int breadth = length * 2;
\\ This is illegal statement because it is using a non-constant value
```

Let's learn to declare Nullable type variables.

Declaring Nullable Type Variables

Nullable types are the instances of **System.Nullable<T>** struct. Nullable structs use predefined value types with a Boolean null indicator. The predefined value type is also called the underlying type. A Boolean variable of a nullable type can be assigned true, false, or null value. You can create nullable types using the ? type modifier. The syntax to declare a nullable type is as follows:

```
System.Nullable<T> variable or T? variable
```

In the preceding syntax, **T** is the underlying value type of the nullable type. **T** can be of any value type, such as **int**, **float**, or **struct**. However, it cannot be a reference type. For instance, **int?** is the nullable type for the type, **int**. In this example, **int** is the underlying type and **?** is the Boolean null indicator. Similarly, **float?** is the nullable type for the value type, **float**. You cannot create nullable type on reference types as they already support null values.

You have learned to declare and assign values to variables and constants, let's learn operators and operator precedence in the following section.

Working with Operators and Operator Precedence

In C#, we can use the value stored inside the variables and constants in the form of an expression. An expression is nothing but a set of language elements, arranged together to perform a specific task or computation. To write expressions, C# 2010 provides a complete set of language elements, such as variables, constants, operators, properties, and literals. A typical example of an expression is shown in the following code snippet:

```
int total= 2 + 3;
```

In the preceding code snippet, expression comprises of an integer variable, **total**, an assignment operator =, integer values 2 and 3, and the addition operator, +. The evaluation of the expression yields the result, 5, which is then assigned to the integer variable **total**.

Operators are nothing but special symbols that are used to specify some computation or other operations, such as arithmetic and logical operations on the operands. Therefore, operators specify the operations to be performed in the expression. The operators in C# 2010 fall under several categories, such as primary, unary, multiplicative, additive, shift, relational and type testing, equality, logical AND, logical XOR, logical OR, conditional AND, conditional OR, null-coalescing, conditional, assignment operators, and lambda expression. Table 2.4 lists different categories and the operators that fall within those categories:

Table 2.4: Operators in C# 2010

Category	Operators	Description
Primary	x.y	Dot operator: Accesses the member variables and methods of a class.
	f(x)	Parenthesis: Specifies the operation order in an expression as well as performs type conversions. It can also be used to invoke a delegate or method.
	a[x]	Square brackets: Specifies that square brackets can be used with arrays, indexers, pointers, and attributes.
	x++	Post-Increment operator: Increases the value of its operand (x) by 1. It first assigns the value of x and then increments its value.
	x--	Post-Decrement operator: Decreases the value of its operand (x) by 1. It first assigns the value of x and then decrements its value.
	new	new operator: Creates objects and calls constructors.
	typeof	typeof operator: Allows you to discover the information about reference and value types using the Type object.
	checked	checked operator: Detects the overflow conditions in integral-type arithmetic operations and conversions.
	unchecked	unchecked operator: Ignores the error caused by the overflow conditions and accepts the generated result, regardless of the overflow conditions.
	->	Pointer operator: Combines pointer dereferencing and member access.

Unary	+	Unary Plus operator: Specifies that this operator is predefined for all numeric types and the result of a unary + operation on a numeric type is always the value of the operand. For example when you execute the following code snippet, the result displayed is 7 as the result of a unary + operation on 7 is always the value of the operand (7):
		`int x =7;`
		`Console.WriteLine(+x);`

	-	Unary minus operator: Specifies that this operator is predefined for all numeric types and the result of a unary – operation on a numeric type is the numeric negation of the operand. For example when you execute the following code snippet, the result displayed is -7 as the result of a unary - operation on 7 is always the numeric negation of the operand (7):
		`int x =7;`
		`Console.WriteLine(-x);`

	!	Logical negation operator: Inverts the result of a Boolean expression.
	~	Bitwise complement operator: Inverts the binary representation of an expression.
	++x	Pre-increment operator: Increases the value of its operand (x) by 1. The pre-increment operation returns the value of the operand (x) after it has been incremented.
	--x	Pre-decrement operator: Decreases the value of its operand (x) by 1. The pre-decrement operation returns the value of the operand (x) after it has been decremented.
	(T)x	Parenthesis: Specifies the operation order in the expression as well as converts the types. It can also be used to invoke a delegate or method.

Table 2.4: Operators in C# 2010		
Category	Operator	Description
	true	true operator: Returns the Boolean value, true when an expression does not satisfy a specified condition
	false	false operator: Returns the Boolean value, false to indicate that an expression does not satisfy a specified condition.
	&	Address operator: Returns the address of its operand.
	sizeof	sizeof operator: Returns the size of value types in bytes.
Multiplicative	*	Multiplication operator: Computes the product of its operands.
	/	Division operator: Computes the division of its operands by dividing the first operand by the second operand.
	%	Modulus operator: Computes the remainder of its operands after dividing the first operand by the second operand.
Additive	+	Binary plus operator: Computes the sum of its operands.
	-	Binary minus operator: Computes the difference of its operands.
Shift	<<	Left shift operator: Shifts its first operand to left by the number of bits specified by the second operand. The number of bits should be of an integer value.
	>>	Right shift operator: Shifts its first operand to right by the number of bits specified by the second operand. The number of bits should be of an integer value.
Relational and type testing	<	Less than operator: Finds out if the first operand is less than the second operand. It returns true if the condition satisfies; otherwise returns false.
	>	Greater than operator: Finds out if the first operand is greater than the second operand. It returns true if the condition satisfies; otherwise returns false.
	<=	Less than equal to operator: Finds out if the first operand is less than or equal to the second operand. It returns true if the condition satisfies; otherwise returns false.
	>=	Greater than equal to operator: Finds out if the first operand is greater than or equal to the second operand. It returns true if the condition satisfies; otherwise returns false.
	is	is operator: Checks the compatibility of a variable with a given type.
	as	as operator: Performs conversions between the compatible reference types.
Equality	==	Equality operator: Checks whether the two expressions are same or not. It returns true if the expressions are equal; otherwise it returns false.
	!=	Inequality operator: Checks whether the two expressions are same or not. It returns true, if the expressions are not equal; otherwise returns false.
Logical AND	&	Logical AND: Computes the logical AND of its Boolean operands. For integral operands, (&), the Logical AND operator compares the corresponding bits of two integrals and computes the logical bitwise AND of the operands. It returns true if and only if both of its operands are true. It can be used as a unary or binary operator.
Logical XOR	^	Logical XOR: Computes the logical exclusive, OR of its Boolean operands. It returns true, if and only if one of its operands is true.

<table>
<tr><th colspan="3">Table 2.4: Operators in C# 2010</th></tr>
<tr><th>Category</th><th>Operators</th><th>Description</th></tr>
<tr><td>Logical OR</td><td>|</td><td>Logical OR: Computes the logical OR of its Boolean operands. For integral operands, | it compares the corresponding bits of two integrals and computes the logical bitwise OR of the operands. It returns true if and only if, any one of its operands is true.</td></tr>
<tr><td>Conditional AND</td><td>&&</td><td>Logical AND: Used to compute the logical AND of its Boolean operands. It evaluates the second operand whenever necessary and returns true, if and only if both its operands are true.</td></tr>
<tr><td>Conditional OR</td><td>||</td><td>Logical OR: Computes the logical OR of its Boolean operands. It evaluates the second operand whenever necessary and returns true, if and only if any of its operands is true.</td></tr>
<tr><td>Null-coalescing</td><td>??</td><td>Null-coalescing operator: Defines a default value for a nullable value type as well as reference type. Returns the left operand, if it is not null; otherwise it returns the right operand.</td></tr>
<tr><td>Conditional</td><td>?:</td><td>Conditional operator: Returns one of the two values based on the result produced by the Boolean expression.</td></tr>
<tr><td rowspan="12">Assignment</td><td>=</td><td>Equal operator: Assigns a value to a variable, property, or an indexer.</td></tr>
<tr><td>+=</td><td>Addition assignment operator: Assigns the result of the sum of the first and second operands to the first operand itself.</td></tr>
<tr><td>-=</td><td>Subtraction assignment operator: Assigns the result of the difference of the first and second operands to the first operand itself.</td></tr>
<tr><td>*=</td><td>Multiplication assignment operator: Assigns the result of the product of the first and second operands to the first operand itself.</td></tr>
<tr><td>/=</td><td>Division assignment operator: Assigns the result of the division of the first operand by second operand to the first operand itself.</td></tr>
<tr><td>%=</td><td>Modulus assignment operator: Assigns the remainder of the division of the first operand by second operand to the first operand itself.</td></tr>
<tr><td>&=</td><td>AND assignment operator: Assigns the logical AND of the first operand and second operand to the first operand itself.</td></tr>
<tr><td>|=</td><td>OR assignment operator: Assigns the logical OR of the first operand and second operand to the first operand itself.</td></tr>
<tr><td>^=</td><td>XOR assignment operator: Assigns the logical XOR of the first operand and second operand to the first operand itself.</td></tr>
<tr><td><<=</td><td>Left-shift assignment operator: Assigns the left shift result of the first operand and second operand to the first operand itself.</td></tr>
<tr><td>>>=</td><td>Right-shift assignment operator: Assigns the right shift result of the first operand and second operand to the first operand itself.</td></tr>
<tr><td>Lambda</td><td>=></td><td>Lambda operator: Separates the input variable on the left side from the statement block or expressions on the right.</td></tr>
</table>

You have learned about operators in C#, next, let's learn to use these operators in an application.

Using Arithmetic Operators

Almost all the computer programs do arithmetic operations, such as addition, subtraction, multiplication, and division. There are five arithmetic operators, plus (+), minus (-), multiplication (*), division (/), and modulus (%) used in C#. The plus (+) sign is used for addition, minus (-) sign is used for subtraction, asterisk (*) is used for multiplication, forward slash (/) is used for division and modulus (%) is used to return the remainder.

Note

You learn about other C# operators in detail in next chapters.

Table 2.5 lists the examples of arithmetic operators:

Table 2.5: Examples of Arithmetic Operators in C#

Operation	Example	Result
Addition	17 + 2	19
Subtraction	9 – 4	5
Multiplication	6 * 5	30
Division	10 / 5	2
Modulus	7 % 5	2

Now, let's consider an example where an assignment statement contains more than one arithmetic operator, as shown in the following code snippet:

```
5 + 3 * 2
```

In the preceding code snippet, answer depends on various things. For example, if you add 5 and 3 first and then multiply the result by 2, the answer is 16. However, if you multiply 3 with 2 first and then add 5 to it, the answer is 11. The possibility of such confusion can be avoided by using the rules of precedence. This means that the arithmetic operations are performed in the specified order of precedence, `multiplication (*), division (/), modulus (%), addition (+), and subtraction (-).`

In an assignment statement, the multiplication and division operations are performed first; then modulus operations, and finally, the addition and subtraction operations. So, the answer of the preceding example would be 11 instead of 16, since the multiplication has been done before the addition.

If the precedence level of the two operators is same (for example the multiplication and division), the operations are performed from left to right, as shown in the following example:

```
20 / 2 * 5
```

In the preceding example, since both the operators have the same precedence level, the division is performed first, yielding the result as 10. Then the multiplication operation is performed, which results in the final answer as 50. If you want to perform multiplication first and then division, then you can use parenthesis so that the operations within the parenthesis can be performed first. You can rewrite the example as follows:

```
20 / (2 * 5)
```

In the preceding example, multiplication is performed first and then the division operation is performed that yields the final result as 2. There are no limitations on the number of parenthesis to be used, but every left parenthesis needs a right parenthesis. If you type an assignment statement in Code Editor with unmatched parenthesis (a syntax error), the end of the statement gets squiggled (underlined text), which indicates an error. When you nest one parenthesis within another parenthesis, the evaluation starts with the innermost set of parenthesis and moves outward, as shown in the following example:

```
((2 + 3) * 5) + 7
```

In this case, the addition of 2 and 3 is performed first; then the result is multiplied by 5; and finally 7 is added to the result. So, the final answer is 32.

Demonstrating Operator Precedence

C# has a large set of operators and you can use many of the operators simultaneously in an expression to perform the required calculations. However, the use of two or more operators may result in the conflict of the operator precedence to decide the operation to be performed first. A C# expression containing operators is evaluated on the basis of the precedence level of each of the operator in the expression. Therefore, the operator precedence refers to a set of rules that specify the order in which the compiler evaluates the expression. When the operators are associated with either the expression on their left or the expression on their right, then it is known as the associativity of operators. Table 2.6 shows the precedence of different operators in C#:

Table 2.6: Operator Precedence

Operator	Associativity
(x), (x.y), f(x), a[x], x++, x--, new, typeof, sizeof, checked, unchecked	Left
+ (unary), -(unary), ~, ++x, --x, (T)x	Left
*, /, %	Left
+ (arithmetic), - (arithmetic)	Left
<<, >>	Left
<, >, <=, >=, is, as	Left
==, !=	Left
&	Left
^	Left
|	Left
&&	Left
||	Left
?:	Right
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	Right

In Table 2.6, the precedence order of C# operators is listed. The operators at the same level have equal precedence and are evaluated in the given order in the expression until and unless they are separated by the parenthesis.

Let's learn to use the scope resolution operator.

Using the Scope Resolution Operator

The :: (scope resolution) operator is added between two identifiers, and the left-hand identifier of the :: operator is taken as a global identifier, or an extern, or alias. When the left-hand identifier of the :: operator is global, the global namespace is searched for the right-hand identifier. Now, let's perform the following steps to create a Console application named **AliasClass**, which shows the use of the :: operator with the global identifier:

1. *Click* Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010, to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** to open the **New Project** dialog box.
3. *Select* **Visual C#** in the **Installed Templates** pane of the **New Project** dialog box and then *select* **Console Application**.
4. *Enter* the name of the application as **AliasClass** in the **Name** text box and *select* the default location, **c:\users\temp\documents\visual studio 2010\Projects** from the **Location** combo box.

5. *Click* the **OK** button. The **AliasClass** application is created.
6. *Add* the highlighted code shown in Listing 2.1 to the **Program.cs** file:

Listing 2.1: Using the Scope Resolution Operator with Global Identifier

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AliasClass
{
    class AliasClass
    {
        public class System
        {
            public void Print()
            {
                global::System.Console.WriteLine("This is an example of :: operator");
            }
        }

        // Define a constant 'Console'
        const string Console = "Hello";
        const string str1 = "World";
        static void Main(string[] args)
        {
            // Error Accesses TestApp.Console
            System sys = new System();
            sys.Print();
            global::System.Console.WriteLine(Console);
            global::System.Console.WriteLine(str1);
            global::System.Console.ReadLine();
        }
    }
}
```

In Listing 2.1, the :: operator is used with the global identifier. This ensures that the types and namespaces are not affected by the inclusion of the new types and members.

7. *Select* **Build→Build Solution** to build the solution, as shown in Fig.C#-2.2:

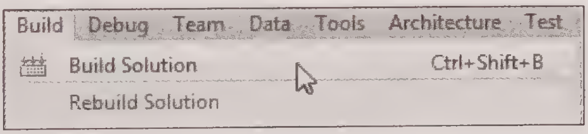


Fig.C#-2.2

8. *Select* **Debug→Start Debugging.** The output appears, as shown in Fig.C#-2.3:

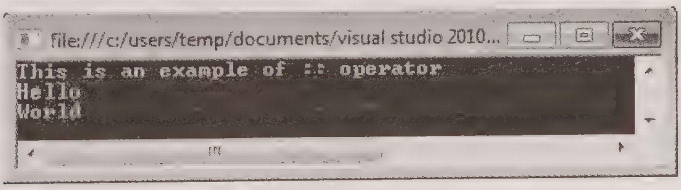


Fig.C#-2.3

In Fig.C#-2.3, the :: operator is used to print the string, Hello World.

Tip

Alternatively, you can press the ***F5*** *key to run an application.*

In the preceding sections, you learned how to work with numeric and boolean variables and constants. Consider a situation where you need a variable to store a word or an entire sentence. The variable types that you have learned so far might not be useful enough to store such types of data. Now, let's learn about the **String** types in C# in the following section.

Working with Strings

A **string** can be thought of as a sequence of Unicode characters encoded in UTF-16. In C#, **string** is an alias for the .NET **String** type that is managed by the .NET Framework. Therefore, **string** and **String** are equivalent and you can use them interchangeably. A **String** is a reference type and represented by double quotes, as shown in the following code snippet:

```
string str = "abc";
```

The **String** class provides various methods to safely perform certain operations on strings, such as string concatenation and string manipulation. Let's discuss how to perform string manipulation in the next section.

String Manipulation

The **System.String** class provides a set of different methods to manipulate strings, such as changing the case of strings and truncating some portions of a string. Let's consider a case, where you have two or more strings and you want the reverse of a particular string literal, or you want to find the length of a string. You can perform all such manipulations on strings by simply creating a string data type and using the dot operator (.) to get the string functionalities. The following code snippet shows an example to manipulate a string:

```
string name = "Visual C# 2010";
string uppername = name.ToUpper();
string lowername = name.ToLower();
string myname = name.ToReplace(name, newname);
```

In the preceding code snippet, variable name is a string variable, which holds the string, **Visual C# 2010**. The **ToUpper()** method turns the case of the value in the **name** variable to uppercase and the **ToLower**() method turns the case of the value in the **name** variable to lowercase. Similarly, the **Replace (name, newname)** method replaces the value of the name variable with the value of the **newname** variable.

String Concatenation

At times, you may need to combine two strings to display a message. This process of adding two strings is called string concatenation. The plus (+) operator is used for concatenation, as shown in the following code snippet:

```
ConcatenatedString = "Visual " + "C# 2010"
```

In the preceding code snippet, the **ConcatenatedString** variable contains the value, **Visual C# 2010**.

Now, let's learn about type safety.

Implementing Type Safety

C# is a type-safe language and the type-safety property of any programming language ensures that a variable can only be accessed by the type associated with that variable. While using the type-safety attribute it is not possible to leave a variable uninitialized, index arrays beyond their bounds, access arbitrary memory locations, or perform illegal type casts. The pre-requisite of a type safe programming language is that you can perform only those operations on a type, which are valid for it. For example, if you pass a floating-point number as a parameter to a method that is defined to accept an integer number as parameter; then, you get a compile-time

error. You can prevent an error in this case by explicitly converting the floating-point number to an integer number. So, it is recommended to use the type-safety feature of C# in programs so that errors can be detected at compile-time itself. You can implement the type-safety attribute in your programs by using generics as well.

Now, let's learn about type conversions.

Type Conversions

C# is a statically typed language; therefore, if you declare a variable as an **int** type, then you cannot assign a **string** type value to it. You can use type conversion to convert the data of one data type into another data type. For instance, there can be times when you need to assign an **int** type value to a variable declared as **float** type. There can also be a situation where you need to pass an integer number to a method that is defined to accept a **double** type value. Such types of operations are known as type conversions.

Type conversion can be of two types, implicit conversion and explicit conversion, as shown in Fig.C#-2.4:

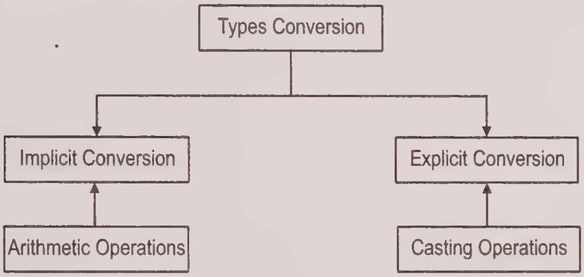


Fig.C#-2.4

In the following section, you learn to implicitly convert a data type into another data type.

Implicit Conversion

If a compiler converts a data type into another data type automatically, it is known as implicit conversion. In implicit type conversion, there is no loss of data. Fig.C#-2.5 illustrates the implicit data conversion:

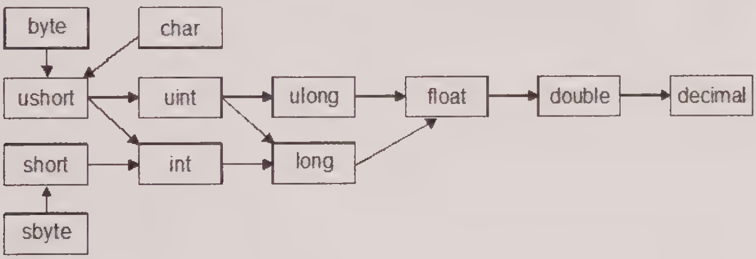


Fig.C#-2.5

In Fig.C#-2.5, the arrows show the direction of the possible conversions with basic data types of C#. Following is an example of implicit conversion:

```
short a = 20;
int b = a;  // Implicit conversion
```

Let's learn about explicit conversion.

Explicit Conversion

When one data type is converted explicitly to another data type, with the help of some pre-defined functions, it is called explicit conversion; and there are the chances of data loss in this process because the conversion is forceful. Some conversions cannot be made implicitly, which are as follows:

- int to short
- int to uint
- uint to int
- float to int

Let's learn about type casting next.

Type Casting

Casting is a process to perform explicit conversion. You can perform casting by prefixing a data type that specifies the type of conversion you want to perform. The syntax for casting is as follows:

```
type variable1 = (type) variable2
```

In the preceding syntax, type in the parenthesis is the conversion type. There may be loss of data in the casting process if you are casting a data type to a smaller data type. The following example shows casting:

```
int x =  (int) 24.46;  // Fractional part of the float value  will be lost i.e. x = 24
```

In the preceding example, the float value 24.46 is converted to integer. Since an integer data type does not allow fractions, the fractional part of the value is truncated and the value, 24 is assigned to variable **x**.

You can convert numeric values into strings in C#, as shown in the following code snippet:

```
int x = 20;
string str = x.ToString();  // x has been converted to string and its value will be as
//string '20' now
```

You can also convert a string into an integer value, as shown in the following code snippet:

```
string str = "20";
int y = int.Parse(str);  //Now the value of y is 20
```

Let's learn about boxing and unboxing.

Boxing and Unboxing

As mentioned earlier, all types whether reference or value types in C# are derived from the **System.Object** base class. Boxing is the process by which you can perform implicit conversion of a value type to an object type. Whenever you perform boxing on a value type, the CLR encloses it inside the **System.Object** base class and stores that value on the managed heap. The process of boxing is explained in the following code snippet:

```
int i= 10;
object ob= (object) i; //Boxing
```

Unboxing on the other hand is the reverse process of boxing. It is the process in which you can perform explicit conversion of an object type to value type. The process of unboxing is shown in the following code snippet:

```
object ob= 123;
int i= (int) ob;
```

Though both boxing and unboxing provides application developers the flexibility to treat value types as object types; computationally, these are quite expensive operations. This is due to the fact that when a value type is boxed, an object needs to be allocated and constructed.

You have learned about data types, variables, strings, and type conversion in C#. Now let's learn to create enumerations.

Creating Enumerations

Enumerations are considered to be the user-defined value types. Allowing you to assign symbolic names to integral values, enums are basically strongly-typed constants. The phrase strongly typed means that the enumeration of one type cannot be assigned to the enumeration of another type, even though the values of their members are same. Enumeration makes it easy to maintain the source code by assigning only the legitimate values. Note that the enumerations are created using the **enum** keyword. The **enum** keyword is an alias for the

System.Enum class. Let's perform the following steps to create an application named **Enumeration**, which shows the use of enumeration:

1. *Repeat* the steps from 1 to 3 of the **Using the Scope Resolution Operator** section of this chapter.
2. *Enter* the name of the application as **Enumeration** in the **Name** text box and then *select* the default location, **c:\users\temp\documents\visual studio 2010\Projects** from the **Location** combo box.
3. *Click* the **OK** button. The **Enumeration** application is created.
4. *Add* the highlighted code shown in Listing 2.2 to the **Program.cs** file:

Listing 2.2: Creating Enumeration

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Enumeration
{
    public enum Color { Red=1, Green, Yellow }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Please select 1 for Red, 2 for Green, and 3 for Yellow");
            string str = Console.ReadLine();
            int colInt = Int32.Parse(str);
            Color col = (Color)colInt;
            switch (col)
            {
                case Color.Red:
                    Console.WriteLine("The selected color is Red");
                    break;
                case Color.Green:
                    Console.WriteLine("The selected color is Green");
                    break;
                case Color.Yellow:
                    Console.WriteLine("The selected color is Yellow");
                    break;
                default:
                    Console.WriteLine("This number is not assigned to any color");
                    break;
            }
            Console.ReadLine();
        }
    }
}
```

Listing 2.2 creates an enumeration named **Color** and defines it using three members named **Red**, **Green**, and **Yellow**. The member **Red** is initialized with **1**; therefore, the value of **Green** and **Yellow** is automatically assigned to **2** and **3** respectively.

5. *Select* **Build→Build Solution** to build the solution.
6. *Press* the **F5** key to execute the application. The output appears, as shown in Fig.C#-2.6:

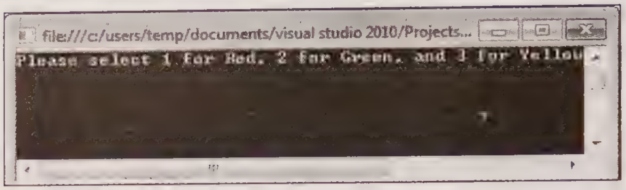


Fig.C#-2.6

7. *Enter* the value **1**, **2**, or **3** on the command prompt. A message appears respective to the entered value, as shown in Fig.C#-2.7:

Fig.C#-2.7

You learned how to store a single value in a variable. In the following section that let's learn about arrays, in which you learn how to use a single variable to store a collection of values of the same data type.

Working with Arrays

An array is a contiguous space in the computer memory that is used to store the values of the variables of same data type. For instance, you can create an array to store 10 integer type values. The values of variables in an array are called the array elements. Array elements are accessed using a variable name and the index number representing the position of the element within the array. In C#, the array index always starts from zero.

The following code snippet shows the declaration of an array:

```
type[] arrayname = new type[size];
```

Now, let's create an array named as **Months** for storing months, as shown in the following code snippet:

```
string []  Months =  new string [12];
```

In the preceding code snippet, an array is created, which has been named as **Months**, of the string type containing 12 (0 to 11) elements.

You can assign values to each of the elements of an array by using the index number (also called the array subscript) of the element. For instance, to assign the value **January** to the first element of an array, you can use the following code snippet:

```
string[] Months = new string[12];
Months[0] = "January";
```

You can also assign values to an array at the time of declaration. However, in case of an explicit initialization, you cannot specify the size of the array, as shown in the following code snippet:

```
string[] Months ={ "January", "February", "March", "April" };
```

An array can be of the following types:

- **Single-Dimensional array:** Consists of only one dimension. For example, int [ ] array1 = new int [5].
- **Multi-Dimensional array:** Contains more than one dimension. For example, int[,] array1 = new int [2, 3].
- **Jagged array:** Refers to a type of array whose elements are also arrays. The elements of a jagged array can be of different dimensions and sizes. In other words, you can call a jagged array as an array of arrays. For instance, int [ ] [ ] array1 = new int [2] [ ] declares a single-dimensional array that has two elements, each of which is a single-dimensional array of integers.

Next, you learn to create single-dimensional and multidimensional arrays.

Creating Single-Dimensional Arrays

A single-dimensional array has only one dimension, which is indexed by a single integer. Now, let's perform the following steps to create an application, named **SingleDimensionArray**, which shows the concept of a single-dimensional array:

1. *Repeat* the steps from 1 to 3 of the **Using the Scope Resolution Operator** section of this chapter.
2. *Enter* the name of the application as **SingleDimensionArray** in the **Name** text box and *select* the default location, **c:\users\temp\documents\visual studio 2010\Projects** from the **Location** combo box.
3. *Click* the **OK** button. This creates the **SingleDimensionArray** application.
4. *Add* the highlighted code shown in Listing 2.3 to the **Program.cs** file:

Listing 2.3: Creating a Single-Dimensional Array

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SingleDimensionArray
{
	class Program
	{
		static void Main(string[] args)
		{
			string[] name = new string[2];
			int[] sub1 = new int[2];
			int[] sub2 = new int[2];
			int[] total=new int[2];
			for (int i=0;i<=1;i++)
			{
				Console.WriteLine("Enter the " + (i + 1) + " student name");
				name[i]=Console.ReadLine();
				Console.WriteLine("Enter the marks of first subject");
				sub1[i] = int.Parse(Console.ReadLine());
				Console.WriteLine("Enter the marks of second subject");
				sub2[i] = int.Parse(Console.ReadLine());
				total[i]=sub1[i]+sub2[i];
			}
			for (int i = 0; i <= 1; i++)
			{
				Console.WriteLine("The student name is " + name[i]
				+ " and his total marks is " +total[i]);
			}
			Console.ReadLine();
		}
	}
}
```

In Listing 2.3, three arrays are created, named **sub1**, **sub2**, and **total**, each having two elements. In addition, we have declared a string array called, **name**, which can contain two elements of type string.

5. *Select* **Build→Build Solution** to build the solution (Fig.C#-2.8).
6. *Press* the **F5** key to execute the application. The output appears, as shown in Fig.C#-2.8:

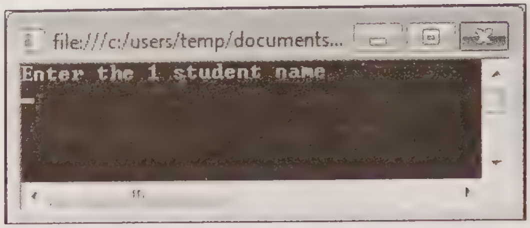


Fig.C#-2.8

7. *Enter* the values for students **1** and **2** respectively on the command prompt. The output appears, as shown in Fig.C#-2.9:

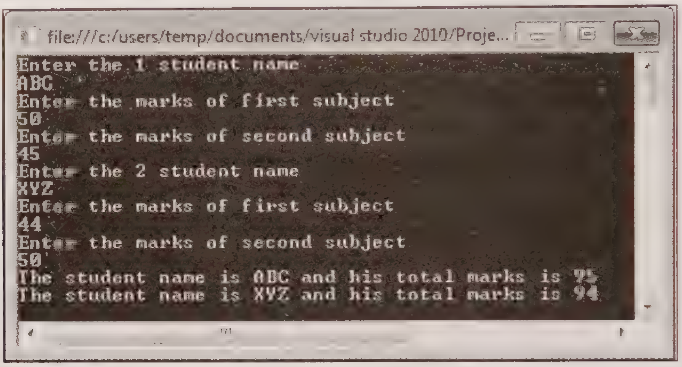


Fig.C#-2.9

Now, let's learn to create a multidimensional array.

Creating Multidimensional Arrays

Multidimensional arrays are indexed by two or more integers. Let's perform the following steps to create an application, named **MultiDimensionArray**, which shows the concept of a multidimensional array:

1. *Repeat* steps 1 to 3 of the **Using the Scope Resolution Operator** section of this chapter.
2. *Enter* the name of the application as **MultiDimensionArray** in the **Name** text box and *select* the default location, **c:\users\temp\documents\visual studio 2010\Projects** from the **Location combo box.**
3. *Click* the **OK** button. This creates the **MultiDimensionArray** application.
4. *Add* the highlighted code shown in Listing 2.4 to the **Program.cs** file:

Listing 2.4: Creating a Multidimensional Array

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MultiDimensionArray
{
    class Program
    {
        static void Main(string[] args)
        {
            int[ , ] x= new int [3,3];//Declaraing Multi-dimensional array
```

```
                for (int j = 0; j <= 2; j++)
                {
                        for (int k = 0; k <= 2; k++)
                        {
                                Console.WriteLine("Enter   the   "   +(k   +   1)
                                 + " element of the " +(j+1)+ " row");
                                x[j,k]=int.Parse(Console.ReadLine());
                        }
                }
                for (int j = 0; j <= 2; j++)
                {
                        Console.WriteLine();
                        for (int k = 0; k <= 2; k++)
                        {

                                Console.Write(x[j,k] +"\t");
                        }
                }
                Console.ReadLine();
        }
    }
}
```

In Listing 2.4, a two-dimensional array, **x** is created, having three rows and three columns.

5. *Select* **Build→Build Solution** to build the solution (Fig.C#-2.10).
6. *Press* the **F5** key to execute the application. The output of the application appears, as shown in Fig.C#-2.10.

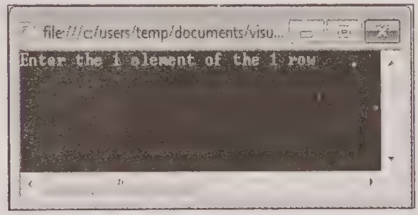


Fig.C#-2.10

7. Enter the appropriate values on the command prompt. The final output appears, as shown in Fig.C#-2.11:

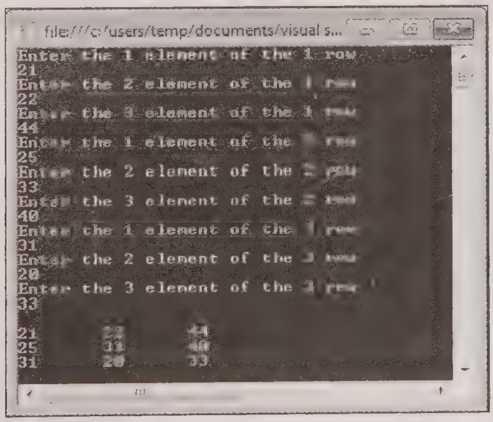


Fig.C#-2.11

Now, let's summarize the key points learned in this chapter.

Summary

In this chapter, you have learned about:

- Relationship between C# and .NET Framework 4.0
- New features introduced in C# 2010, such as dynamic types, named and optional arguments, enhanced COM interoperability, covariance and contravariance, new command-line compiler options, and implicit line continuations
- Different types of keywords in C# 2010 such as reserved keywords and contextual keywords
- Different data types in C# 2010, such as value types, reference types, and pointer types
- Variables and constants and how these can be declared and used in an application
- Operators and the way these are used in an application
- The conversion of one data type to another by using type conversion, casting techniques, and boxing and unboxing
- Enumerations and how to create them
- Arrays and their different types, such as single-dimensional, multidimensional, and jagged arrays

Chapter 3

Working with Control Statements and Exception Handling

In this Chapter:

- Working with Statements
- Working with Selection Statements
- Exploring Loops or Iteration Statements
- Exploring Jump Statements
- Working with Exception
- Commenting a C# Program

The flow of a program written in any programming language implies the way in which the program executes. Normally, a program written in C# executes in a sequential manner. In case you want to change the flow of execution of the program, you need to use the control flow statements, such as if else and switch statements. The control flow statements change the flow of execution of a program according to your requirements. For instance, you can use the control flow statements to execute only specific blocks of code on the fulfillment of specific conditions. You can also use the control flow statements to transfer the program control from one block of code to another.

Sometimes a situation may occur when your program is logically correct and compiles successfully, but it fails to display the required output and terminates abruptly. Such kind of situations may arise due to an exception in your program. An abnormal condition that often occurs while a program is being executed and disrupts the normal flow of application is known as an exception.

In this chapter, you learn how to work with the different statements used in C#, such as selection statements, iteration statements, and jump statements. You also learn about exceptions and how to handle them. In the end, you learn about comments, their types, how to addadding comments to a C# program.

Let's start by learning how to work with different statements in C#.

Working with Statements

In C#, statements serve as building blocks for any program. You can perform varied functions with the help of statements, such as declaring a local variable (a variable which is defined inside a method and has no presence outside the method), calling a method, and creating an object. In C#, some of the most commonly used statements are as follows:

- **Simple statement:** Consists of a single line of code. It is terminated by a semicolon. The following code snippet shows an example of a simple statement:

```
int var1 = 5;
```

In the preceding code snippet, we have defined a variable named var1 and initialized it with value 5.

- **Compound statement:** Consists of more than one statements enclosed in braces. A compound statement is sometimes also called a block statement and usually appears as the body of another statement, such as the if statement.

- **Empty statement:** Specifies that no operations are required to be performed but a statement is required. You can specify an empty statement by a single semicolon. For example, the following code snippet shows the use of the empty statement:

```
void login() {
    if(x > 3) goto exit;
    // ...
    exit: ;
}
```

  In the preceding code snippet, when the **login()** method is called, the condition in the if statement is checked. Suppose the **x** variable stores the number of login attempts by a user, if it exceeds 3 then the program control is transferred to exit. In such a case, a statement is required but no action is required to be performed and therefore an empty statement is used.

- **Expression statement:** Evaluates an expression and stores the resulting value in a variable. An expression statement is shown in the following code snippet:

```
float volume = length * breadth * height; //assigns the volume of a cuboid to the
                                          //float variable volume
```

- **Selection statement:** Enables you to control the flow of program execution depending on one or more conditions or criteria. The **if...else** and **switch** statements are the examples of selection statements used in C#.

- **Iteration statement:** Allows you to repeatedly perform a specified task(s) until certain condition is reached. The examples of iteration statements are **for**, **while**, **do...while**, and **for each** loops.
- **Jump statement:** Transfers the program control to a different section of code. The **break**, **continue**, **goto**, **default**, and **return** statements are examples of the jump statements.

Note

You learn about the selection statements, iteration statements, and jump statements in detail later in this chapter.

Let's now learn about the selection statements.

Working with Selection Statements

In C#, the statements that cause the flow of a program to change based on a certain condition are known as selection statements. The program control checks a condition before executing the code inside the selection statement. The selection statement causes the program control to adopt a specific flow of execution based on whether or not a certain condition is true. C# supports the following two types of selection statements:

- **if statement:** Allows you to test whether or not a certain condition is true. If the condition is true, the program control is transferred to the block of code that is inside the **if** statement; otherwise, the program control is transferred to another block of code.
- **switch statement:** Allows you to compare an expression with different values.

Now, let's discuss each of these selection statements in detail.

The if Statement

One of the most important and commonly used selection statement in C# is the **if** statement. The **if** statement evaluates a specified condition or a set of conditions to true or false. If the condition is true, the program control enters the **if** block and executes all the statements and instructions inside the **if** block. The syntax of the **if** statement is as follows:

```
if condition  // condition {
   //statement
}
```

In the preceding syntax, if the condition inside the **if** block is evaluated to true, then the program control is transferred to the statements inside the **if** block and executes the statements.

Now, let's create a console application named **If Example** to learn how to use the **if** statement. Perform the following steps to create the **If Example** application:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 integrated development environment (IDE).
2. *Select* **File→New→Project** from the menu bar.

   The **New Project** dialog box appears.
3. *Select* **Visual C#** from the **Installed Templates** pane and then select the **Console Application** option from the middle pane of the **New Project** dialog box.
4. *Enter* **If Example** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have selected the location, **D:\C#2010 Applications**.
5. *Click* the **OK** button. The **If Example** application is created, as shown in Fig.C#-3.1:

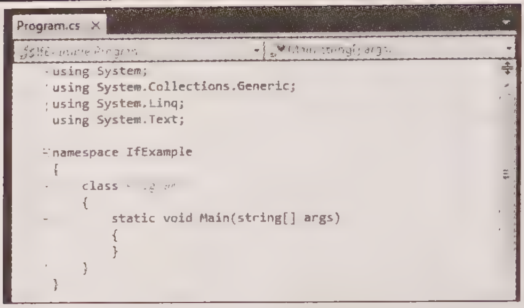


Fig.C#-3.1

As you can see in Fig.C#-3.1, whenever you create a console application in Visual Studio 2010, the Visual Studio IDE automatically generates some files for your application. The first is the **Properties** file, which contains the configuration security, and deployment options that you can apply to your entire console application. For example, you can set the properties for your entire application in the **Properties** file to indicate whether a file should be compiled or embedded into the build output as a resource. The second file is the **References** file that identifies a binary file, which is required to run the application. In other words, it contains the references of all the assemblies that are required by your application. The third file is the **Program.cs** file, which is the source code file where you actually write the code for an application. The **Program.cs** file contains the entry point for an application.

6. *Add* the highlighted code given in Listing 3.1 in the **Program.cs** file of the **If Example** application:

Listing 3.1: Using the if Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace IfExample
{
  class Program
  {
        static void Main(string[] args)
        {
                int num;
                Console.WriteLine("Please enter a number:");
                num = Convert.ToInt32(Console.ReadLine());
                if (num % 2 == 0)
                {
                    Console.WriteLine("The number entered is even");
                }
                Console.ReadLine();
        }
  }
}
```

In Listing 3.1, the Console.ReadLine() method is used to take input from a user, which is stored in the variable, **num**. The **if** statement checks the number entered by the user. If the number is divisible by 2, then the statement inside the if block is executed. In case, the user enters a number that is not divisible by two; no output is displayed, because there is no condition set for this.

7. *Press* the **F5** key to run the **If Example** application. The output appears, as shown in Fig.C#-3.2:

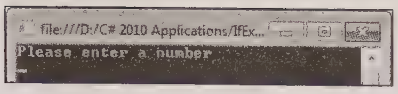


Fig.C#-3.2

As seen in Fig.C#-3.2, you are prompted to **enter** a number.

8 *Enter* any number. In our case, we have entered the number **120**, as shown in Fig.C#-3.3:

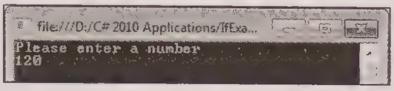


Fig.C#-3.3

9 *Press* the **Enter** key. The output appears, as shown in Fig.C#-3.4:

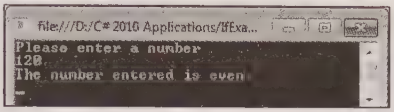


Fig.C#-3.4

After you enter a number, the **if** statement inside the program checks the number. In case, the number is divisible by two, the program control enters the **if** block and displays the result that the number entered is even.

However, note that we have not specified any code that must execute when you enter a number that is not divisible by 2. For this purpose, an **else** clause is used with the **if** statement. The **else** clause is executed when the condition for the **if** block is not fulfilled. In other words, when the condition for the **if** block evaluates to **false**, the block of code inside the **else** block is executed.

The syntax of the **if…else** statement is as follows:

```
if condition  // condition
{
   //statement
}
else
{
   // else statement
}
```

In the preceding syntax, if the condition specified inside the **if** block is evaluated to false, the statement inside the **else** block is executed.

Now, let's consider the **If Example** application created in Listing 3.1 to learn how to use the **if…else** statement. Perform the following steps in the **If Example** application to learn how an if…else statement works:

1 *Add* the code, given in Listing 3.2, after the code of the **if** block given in Listing 3.1:

Listing 3.2: Using the else Statement

```
//This is the else block.
   else
        {
              Console.WriteLine("The number entered is odd");
        }
```

2 *Press* the **F5** key to run the **If Example** application and enter an odd number. The output appears, as shown in Fig.C#-3.5:

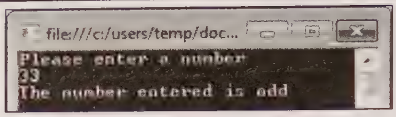


Fig.C#-3.5

As seen in Fig.C#-3.5, you are prompted to enter a number. The **if** statement checks whether the number you enter is divisible by 2 or not. As the number entered in our case is not divisible by 2; therefore, the code within the **else** block executes and the message, The number entered is odd, appears on the screen. Now, let's learn how to nest multiple **if** statements.

By nesting multiple **if** statements, you can further refine the process of taking decisions on the basis of the specific condition, as it helps in examining multiple conditions. As a program increases in complexity, the number of conditions that you use to execute the code also increases. In such cases, it is better to nest multiple **if** statements to attain the required result. The syntax of using the nested **if** statements is as follows:

```
if condition  //condition
{
  //statement
}
else
if condition  //condition
{
  //statement
}
else
{
  //statement
}
```

In the preceding syntax, if the condition in the first **if** block is evaluated to true, then the first **if** block is executed; otherwise, the program control transfers to the **else...if** block and checks the condition specified in the if statement. Now, if this condition evaluates to true, then the block of code inside the **else...if** block executes; otherwise, the program control transfers to the **else** block.

Now, let's create an application named **NestedIf** to demonstrate the use of the nested **if** statement. Perform the following steps to create the **NestedIf application**:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **If Example**.application.
2. *Enter* **NestedIf** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have entered, **D:\C# 2010 Applications**, as the location.
3. *Click* the **OK** button. The **NestedIf** application is created.
4. *Add* the highlighted code given in Listing 3.3 in the **Program.cs** file:

Listing 3.3: Using the Nested if Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace NestedIf
{
  class Program
  {
        static void Main(string[] args)
        {
               Console.WriteLine("Finding the Largest of Three Numbers\n");
```

```
            Console.Write("Enter the First Number:");
            int number1 = int.Parse(Console.ReadLine());
            Console.Write("Enter the Second Number:");
            int number2 = int.Parse(Console.ReadLine());
            Console.Write("Enter the Third Number:");
            int number3 = int.Parse(Console.ReadLine());
            if (number1 > number2 && number1 > number3)
            Console.WriteLine(number1 + " is the Largest Number");
            else
            if (number2 > number3)
                    Console.WriteLine(number2 + " is the Largest Number");
            else
            Console.WriteLine(number3 + " is the Largest Number");
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
}
```

In Listing 3.3, three integer variables, **number1**, **number2**, and **number3**, have been declared. These variables are used to store three integer numbers that are accepted as input from the user. The first **if** statement checks if the number stored in the variable, **number1** is greater than the numbers stored in the variables, **number2** and **number3**, respectively. If this condition is true then the **Console.WriteLine()** statement prints the message that **number1** is the largest number. However, if the condition specified in the **if** statement evaluates to **false**, then the program control enters the **else** block. The **if** statement inside the **else** block checks whether the number stored in the **number2** variable is greater than the number stored in the **number3** variable. If this condition evaluates to **true** then the **Console.WriteLine()** statement inside the **if** statement executes and prints the message that **number2** is the largest number. In case, the number stored in **number2** variable is smaller than the number stored in the **number3** variable, then the program control enters the **else** block. The **Console.WriteLine()** statement inside the **else** block executes and prints the message that **number3** is the largest number.

5 *Press* the **F5** key to run the **NestedIf** application.

The output appears, as shown in Fig.C#-3.6:

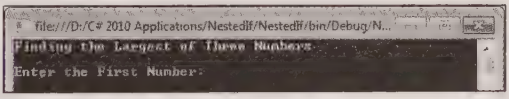


Fig.C#-3.6

As seen in Fig.C#-3.6, you are prompted to enter the first number. When you have entered the first number, press the **ENTER** key and you are prompted again to enter the second number, as shown in Fig.C#-3.7:

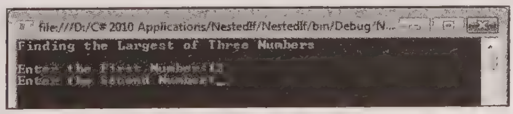


Fig.C#-3.7

Similarly, you are prompted to enter the third number. As soon as you enter all the three numbers, the **if...else** block checks them and returns the largest of the three numbers, as shown in Fig.C#-3.8:

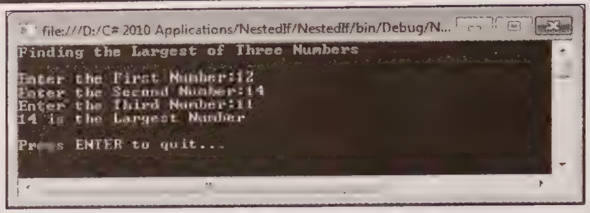


Fig.C#-3.8

Now, let's learn about the switch statement.

The Switch Statement

Another form of selection statement in C# is the **switch** statement, which is an alternative for making selections in an easier way. Unlike the **if** statement that can be used to check only a limited number of conditions, a **switch** statement allows you to provide a list of possible values for a variable. You should use the **switch** statement with the **case** statement and there can be any number of **case** statements inside the **switch** statement. The **switch** statement passes the program control to one of the **case** statements whose value matches with that of the variable specified in the **switch** statement. The syntax of the **switch** statement is as follows:

```
switch (variablename) // test expression
{
  case 1://constant-expression
{
  //satement
  }
  break;
  case 2://constant-expression
  {
       //satement
  }
  break;
  default:
  break;
}
```

In the preceding syntax, the **switch** statement compares a test expression with different **case** statements. The program control passes from one **case** to another until a **case** statement is found whose constant-expression matches with that of the test expression. The statements within that **case** statement are then executed.

Now, let's create a console application named **SwitchStatement** to demonstrate how to use the **switch** statement. Perform the following steps to create the **SwitchStatement** application:

1. *Repeat* steps **1** to **3** as discussed earlier while creating the **If Example** application in the **The if Statement** section of this chapter.
2. *Enter* **SwitchStatement** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have entered the location as D:\C# 2010 Applications.
3. *Click* the **OK** button. The **SwitchStatement** application is created.
4. *Add* the highlighted code given in Listing 3.4 in the **Program.cs** file:

Listing 3.4: Using the switch Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace SwicthStatement
```

```
{
  class Program
  {
        static void Main(string[] args)
        {
                string Name;
                int Marks;
                Console.Write("Enter Your Name:");
                Name = Console.ReadLine();
                Console.WriteLine("Select the Range of Marks:");
                Console.WriteLine("1.   0-50");
                Console.WriteLine("2.   51-100");
                Console.Write("Enter Your Choice (1,2):");
                Marks = int.Parse(Console.ReadLine());
                switch (Marks)
                {
                        case 1:
                        {
                                Console.Write(Name + " your marks are Low.\n");
                        }
                        break;
                        case 2:
                        {
                                Console.Write(Name + " your marks are Good. \n");
                        }
                        break;
                        default:
                        break;
                }
                Console.Write("\nPress ENTER to quit...");
                Console.ReadLine();
        }
  }
}
```

In Listing 3.4, a string variable called **Name** and an integer variable called **Marks** are declared. The **Name** variable is used to store the name of the user while the **Marks** variable is used to store either of the two options, **1** and **2** that specify the range within which your marks lie. The value stored in **Marks** is then matched with the value in the **case** statement. The code inside the **case** statement whose value matches with the value of the **Marks** variable is executed.

5. *Press* the **F5** key to run the **SwitchStatement** application.

   The output appears, as shown in Fig.C#-3.9:

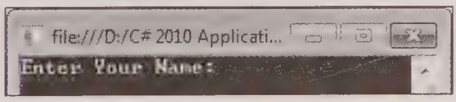


Fig.C#-3.9

As seen in Fig.C#-3.9, you are prompted to enter your name.

6. *Enter* your name and *press* the **Enter** key.

   Now, you are prompted to select the range in which your marks lie, as shown in Fig.C#-3.10:

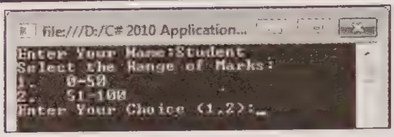


Fig.C#-3.10

7 *Enter* your choice and *press* the **Enter** key. In our case, we have entered **2**, as shown in Fig.C#-3.11:

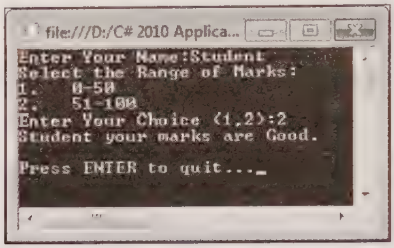


Fig.C#-3.11

In Fig.C#-3.11, when you select the range of marks, the program control is transferred to the **case** block that matches the range, and that **case** block is executed to produce the result.

Now, let's learn about the different iteration statements or loops used in C#.

Exploring Loops or Iteration Statements

In C#, you can use iteration statements to create loops and execute the code for a given number of times. For example, consider a situation in which you want to execute the same set of code 50 times in your program. In such a situation, instead of writing the code 50 times, you can define the code in a loop and execute the loop 50 times. This saves the complexity and time involved in coding. An iteration statement or a loop executes a statement or a set of statements in a repeated manner. The following are the four types of iteration statements used in C#:

- The **while** loop
- The **do...while** loop
- The **for** loop
- The **foreach** loop

Now, let's discuss these in detail, one by one.

The while Loop

You can use the **while** loop to execute a statement or block of statements until the specified condition evaluates to false. The best situation to use the **while** loop in your program is when you do not know in advance how many time the loop needs to be executed. The condition in the **while** loop is evaluated before executing the loop and the statement(s) inside the **while** loop are executed repeatedly as long as the condition evaluates to true. This implies that if the condition in the **while** loop evaluates to true, then the statements inside the **while** loop are executed; otherwise, the statements are not executed. When the statements inside the **while** loop are executed, the program control is transferred to the starting point of the **while** loop to check the condition again for the next iteration. The syntax of the **while** loop is as follows:

```
while (condition)        //condition
{
  //statement
}
```

Now, let's create a console application named **WhileLoop** to demonstrate how to use the **while** loop. Perform the following steps to create the **WhileLoop** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **If Example** application in the **The if Statement** section of this chapter.
2. *Enter* **WhileLoop** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have retained the default location, which is C:\users\temp\documents\visual studio 2010\Projects.
3. *Click* the **OK** button. The **WhileLoop** application is created.
4. *Add* the highlighted code given in Listing 3.5 in the **Program.cs** file:

Listing 3.5: Using the while Loop

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace WhileLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Loop 1:");
            int i = 0;
            while (i < 5)
            {
                Console.WriteLine("I = {0}", i);
                i++;
            }
            Console.ReadLine();
        }
    }
}
```

In Listing 3.5, the **while** loop is used to execute the code inside it repeatedly till the specified condition, that is, **i<5** evaluates to true. The condition in the **while** loop is checked before the while loop is executed. If the condition evaluates to **true**, the statements inside the **while** loop are executed.

5. *Press* the **F5** key to run the **WhileLoop** application. The output appears, as shown in Fig.C#-3.12:

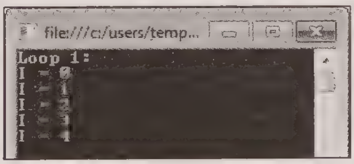


Fig.C#-3.12

As you can see in Fig.C#-3.12, the while loop continues to print the incremented value of the variable, i, until it remains less than 5.

The do...while Loop

The function of the **do...while** loop is similar to that of the **while** loop. The primary difference between the **do...while** and **while** loops is that the condition in the **do...while** loop is checked at the end of the loop. This implies that the **do...while** loop executes at least once even if the condition evaluates to false. The syntax of the **do...while** loop is as follows:

```
do
{
 //statement
  }
while (condition); //condition
```

Now, let's create a console application named **DoWhileLoop** to demonstrate how to use the **do...while** loop. Perform the following steps to create the **DoWhileLoop** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **If Example** application in the **The if Statement** section of this chapter.
2. *Enter* **DoWhileLoop** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have retained the default location which is C:\users\temp\documents\visual studio 2010\Projects.
3. *Click* the OK button. The **DoWhileLoop** application is created.
4. *Add* the highlighted code given in Listing 3.6 in the **Program.cs** file:

Listing 3.6: Using the do…while Loop

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace DoWhileLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 5;
            Console.WriteLine("Loop 1:");
            do
            {
                Console.WriteLine("I = {0}", i);
                i++;
            }
            while (i < 20);
            Console.ReadLine();
        }
    }
}
```

In Listing 3.6, an integer variable, **i**, is declared and assigned the value 5. The **do...while** loop generates numbers from 5 to 19. The condition, **i<20**, in the **do…while** loop is checked at the end of the loop. When this condition evaluates to false, the **do...while** loop is terminated.

5. *Press* the **F5** key to run the application. The output of the application appears, as shown in Fig.C#-3.13:

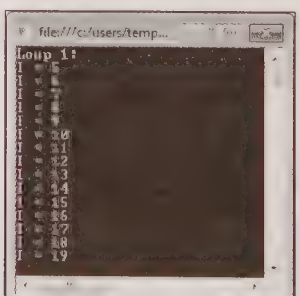


Fig.C#-3.13

Next, let's discuss *about* the for loop.

The for Loop

The function of the **for** loop is also similar to that of the **while** loop. The only difference between the two is in the syntax. The syntax of the **for** loop includes an **initializer**, **condition**, and a **loop expression**. The initializer initializes a variable, the **loop expression** increments or decrements the value of the variable, and **condition** refers to a condition specified in the for loop. A for loop repeatedly executes the statement(s) enclosed within it until the specified condition evaluates to false. You can use the **for** loop when you know how many time you want to execute a statement or a block of statements. The syntax of the **for** loop is as follows:

```
for ( initializer; condition; loop expression)
{
...
...      //Statements
}
```

In the preceding syntax, **initializer** is the starting value of a variable, **condition** is the expression that is checked before the execution of a loop, and **loop expression** either increments or decrements the loop counter.

Now, let's create a console application named **ForLoop** to demonstrate the use of the **for** loop. Perform the following steps to create the **ForLoop** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **If Example** application in the **The if Statement** section of this chapter.
2. *Enter* **ForLoop** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have selected the default location, which is **C:\users\temp\documents\visual studio 2010\Projects**.
3. *Click* the **OK** button. The **ForLoop** application is created.
4. *Add* the highlighted code given in Listing 3.7 in the **Program.cs** file:

Listing 3.7: Using the for Loop

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ForLoop
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 5; ++i)
                Console.WriteLine("Welcome to C# 2010 in Simple Steps.");
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
}
```

In Listing 3.7, an integer variable, **i**, is declared and initialized to **0** inside the **for** loop. The condition for the **for** loop is set to **i<5**. In addition, the **i** variable is incremented by **1** in the for loop expression.

5. *Press* the **F5** key to run the **ForLoop** application. The output appears, as shown in Fig.C#-3.14:

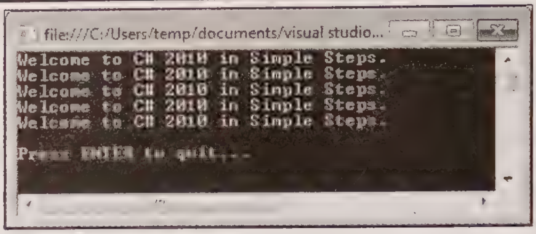


Fig.C#-3.14

In Fig.C#-3.14, the **for** loop displays the statement, **Welcome to C# 2010 in Simple Steps.**, for a fixed number of times. This is an example of using the **for** loop for a fixed number of times.

Note

You can also create an infinite for loop. For creating an infinite for loop, you do not need to specify the loop expression. This implies that you need not to specify the value with which you want to increment or decrement the value of a variable in the for loop.

Next, let's discuss about the **foreach** loop.

The foreach Loop

The **foreach** loop iterates through all the items in a collection, such as an array, which is an organized collection of variables of the same data type. The working of **foreach** loop is similar to that of the **for** loop. The statements in the **foreach** loop continue to execute for each element in a collection. After the **foreach** loop completes iterating through all the elements in a collection, the program control transfers to the next block of code. The syntax of the **foreach** loop is as follows:

```
foreach(<type> <item name> in <list>)
{
  ...
  ...   //Statements
}
```

In the preceding syntax, **type** refers to the type of item contained in the list, **item name** is the meaningful name for a variable, **in** is a keyword, and **list** is the name of the collection of objects referred to in the **foreach** loop.

Now, let's create a console application named **ForEachLoop** to demonstrate the use of the **foreach** loop. Perform the following steps to create the **ForEachLoop** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **If Example** application in the **The if Statement** section of this chapter.
2. *Enter* **ForEachLoop** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have selected the default location, which is C:\users\temp\documents\visual studio 2010\Projects.
3. *Click* the **OK** button. The **ForEachLoop** application is created.
4. *Add* the highlighted code given in Listing 3.8 in the **Program.cs** file:

Listing 3.8: Using the foreach Loop

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ForEachLoop
{
  class Program
```

```
    {
        static void Main(string[] args)
        {
            string[] bookTitles = new string[] { "Alice in Wonderland",
            "Pelican Brief",
            "Ignited Minds" };
            Console.WriteLine("The available book titles are: ");
            foreach (string i in bookTitles)
            Console.WriteLine("\t" + i);
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
}
```

In Listing 3.8, the **foreach** loop iterates through all the items in the string array, **book Titles**, and displays the result in the output window.

5. *Press* the **F5** key to run the **ForEachLoop** application. The output appears, as shown in Fig.C#-3.15:

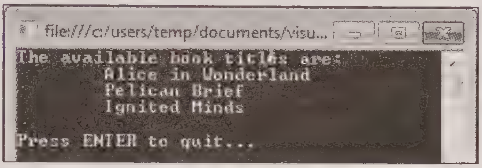


Fig.C#-3.15

Now, let's discuss about different jump statements in C#.

Exploring Jump Statements

The jump statements allow you to move the program control from one point to another at any particular instance during the execution of the program. The jump statements used in C# are **break, continue, default, goto,** and **return**. In this chapter, we discuss only the first two jump statements - **break** and **continue**, in the following sections.

The break Statement

The **break** statement terminates the loop in which it exists. It also changes the flow of the execution of a program. After the loop is terminated, the program control is transferred to the next block of statements following the **break** statement. For instance, you can use the **break** statement within a **switch** statement to exit a **case** block. You can also use the **break** statement with the iteration statements.

Now, let's create a console application named **BreakStatement** to demonstrate the use of the **break** statement. Perform the following steps to create the **BreakStatement** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **IfExample** application in the **The if Statement** section of this chapter.
2. *Enter* **BreakStatement** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have retained the default location, which is C:\users\temp\documents\visual studio 2010\Projects.
3. *Click* the **OK** button. The **BreakStatement** application is created.
4. *Add* the highlighted code given in Listing 3.9 in the **Program.cs** file:

Listing 3.9: Using the break Statement

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace BreakStatement
{
    class Program
    {
        static void Main(string[] args)
        {
            for (int i = 0; i < 10; ++i)
            {
                if (i == 7)
                break;
                Console.WriteLine("Number {0}", i);
            }
            Console.ReadLine();
            Console.Write("\nPress ENTER to quit...");
        }
    }
}
```

In Listing 3.9, the **break** statement terminates the **for** loop when the value of **i** becomes **7**.

5. *Press* the **F5** key to run the **BreakStatement** application. The output appears, as shown in Fig.C#-3.16:

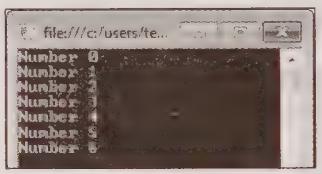


Fig.C#-3.16

Now let's learn about the **continue** statement.

The continue Statement

The **continue** statement is used to transfer the program control to the beginning of the loop till a specific condition is satisfied. When the specified condition evaluates to false, the **continue** statement transfers the program control to the next statement.

Now, let's create a console application named **ContinueStatement** to demonstrate the use of the **continue** statement. Perform the following steps to create the **ContinueStatement** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **IfExample** application in the **The if Statement** section of this chapter.
2. *Enter* **ContinueStatement** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have retained the default location, which is C:\users\temp\documents\visual studio 2010\Projects.
3. *Click* the **OK** button. The **ContinueStatement** application is created.
4. *Add* the highlighted code given in Listing 3.10 in the **Program.cs** file:

Listing 3.10: Using the continue Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ContinueStatement
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            int number = 1;
            while (true)
            {
                Console.WriteLine(number);
                number++;
                if (number <= 10)
                    continue;
                else
                break;
            }
            Console.WriteLine("Series is broken at number: " + number);
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();

        }
    }
}
```

In Listing 3.10, the **while** loop will iterate infinitely, until the **break** keyword is executed. Inside the **while** loop, the **if** statement checks the condition whether the value of the variable is less than or equal to 10, and if this condition is evaluated to true, then the program control is transferred to the **continue** statement. The **continue** statement, in turn, transfers the program control to the beginning of the loop as long as the condition is satisfied. If the condition is evaluated to false, the program control is transferred to the **else** block and the execution of the **while** loop terminates. The **break** keyword inside the **else** block causes the program to terminate.

5 *Press* the **F5** key to run the **ContinueStatement** application. The output of the application appears, as shown in Fig.C#-3.17:

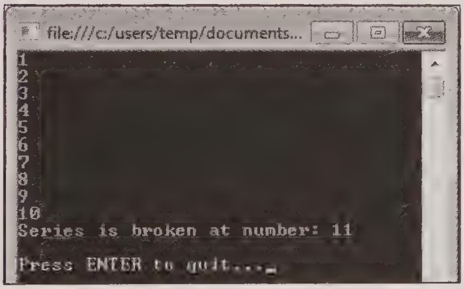


Fig.C#-3.17

Let's now learn about exceptions.

Working with Exceptions

An exception is a runtime error that arises because of some unexpected situation. The exception handling feature in C# helps you to deal with any unexpected situation or abnormal condition that might occur when a program is being executed. For example, dividing a number by zero or passing a string value to an integer variable. During the execution of a program, if an exception occurs you can terminate the program explicitly by using the **throw** statement.

Now, let's first learn about the types of exceptions in C#.

Describing Types of Exceptions

As mentioned earlier, an exception can be thought of as a runtime anomaly, error condition, or some unexpected behavior of a program that is encountered during its execution. Exceptions usually arise due to some fault in the source code or because of an attempt to access an unavailable resource, such as a file that has been removed from the memory. You can handle exceptions by using the exception handling mechanism. This mechanism helps you to detect and handle errors that occur at run time. Though the exception handling mechanism does not guarantee that the code you develop will be completely bug-free, yet upto certain extent, it helps to build applications that do not behave unexpectedly during execution.

All exceptions in the .NET Framework are derived from the **Exception** class. In C#, exceptions are usually of the following two types:

- System-Defined Exceptions
- User-Defined Exceptions

Let's now discuss these two types of exceptions one by one.

System-Defined Exceptions

The exceptions defined by the .NET Framework Base Class Library (BCL) are often referred to as **system-defined exceptions**. The system-defined exceptions are also known as **predefined exceptions** and are represented by their respective exception classes. Some of the most common exception classes defined in the .NET Framework BCL are as follows:

- System.ArithmeticException
- System.ArrayTypeMismatchException
- System.DivideByZeroException
- System.IndexOutOfRangeException
- System.NullReferenceException
- System.OutOfMemoryException
- System.OverflowException
- System.StackOverflowException
- System.TypeInitializationException
- System.InvalidCastException

All the predefined exception classes ultimately inherit from the **System.Exception** base class.

User-Defined Exceptions

In C#, you can also create your own exceptions, which are known as user-defined exceptions. These exceptions are also sometimes referred to as custom exceptions. As mentioned before, all the pre-defined exceptions are handled by the classes that derive from the **System.Exception** class. However to define user-defined exceptions you must create classes that derive from the **System.ApplicationException** class. The syntax to create a user-defined exception is as follows:

```
class MyUserDefinedException : ApplicationException
{
 //some code to handle the exception
}
```

You can define a user-defined exception in the Program.cs file as follows:

```
public class StudentDataNotFoundException : ApplicationException
{
  public StudentDataNotFoundException (string exMessage) : base (exMessage)
  {
```

```
            //some code
        }
    }
```

Let's now discuss how to handle exceptions.

Handling Exceptions

You can use the exception handling features provided in the .NET Framework to deal with error conditions that often result into exceptions. In C#, we generally use the **try**, **catch**, and **finally** keywords to monitor any code for exceptions, handle exceptions, and perform cleanup operations, such as closing a database connection. Now, let's discuss the **try**, **catch**, and **finally** statements individually in the next section.

The try…catch…finally Statement

In C#, you can handle exceptions by using three statements - **try, catch,** and **finally**. The **try** block consists of the **try** statement with an opening curly brace that indicates the start of the **try** block followed by the block of statement(s) that can raise an exception. Finally, a closing curly brace marks the end of the **try** block. The **catch** block immediately follows the **try** block and is used to enclose the code that executes when an exception occurs in the **try** block. One or more **catch** blocks can exist for a **try** block. If an exception is raised in the **try** block then the program control is transferred to the appropriate **catch** block and the code in the **catch** block is executed. Irrespective of whether or not an exception has occurred, the **finally** block always executes. The **finally** block is generally used to perform a clean-up process. If any exception is raised in the **try** block, the program control is directly transferred to its corresponding **catch** block and later to the **finally** block. If no exception occurs inside the **try** block, then the program control is transferred directly to the **finally** block.

Now, let's create a console application named **TryStatement** to demonstrate the use of the **try…catch…finally** statement. Perform the following steps to create the **TryStatement** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **IfExample** application in the **The if Statement** section of this chapter.
2. *Enter* **TryStatement** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have retained the default location, which is **C:\users\temp\documents\visual studio 2010\Projects**.
3. *Click* the **OK** button. The **TryStatement** application is created.
4. *Add* the highlighted code given in Listing 3.11 in the **Program.cs** file:

Listing 3.11: Using the try…catch…finally Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace TryStatement
{
  class Program
  {
       static void Main(string[] args)
       {
             int x = 0;
             int div = 0;
             try
             {
                   div = 100 / x;
                   Console.WriteLine(div);
             }
             catch (DivideByZeroException de)
             {
                   Console.WriteLine("Exception Occured");
             }
             finally
```

```
                {
                        Console.WriteLine("Executing Finally Block");
                }
                Console.ReadLine();
            }
        }
    }
```

In Listing 3.11, two integer variables, **x** and **div**, are declared and assigned the value **0**. The **try** block encloses the statement, **div=100/x**, which can raise an exception. When an exception is raised the program control is transferred to the **catch** block, which executes the **Console.WriteLine()** statement that it encloses. In the end, the program control is transferred to the **finally** block, which executes the **Console.WriteLine()** statement.

5. *Press* the **F5** key to run the application. The output of the application appears, as shown in Fig.C#-3.18:

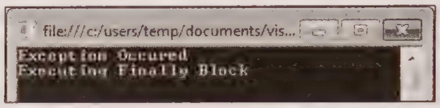


Fig.C#-3.18

Now, let's learn about the throw statement in the next section.

The throw Statement

The **throw** statement in C# signifies that an exception has been occurred during the execution of a program. During the execution of the program, if the program encounters a **throw** statement, the program terminates and returns the error. The most common way of using the **throw** statement is to use it with the **try...catch...finally** statement. When an exception is thrown, the program looks for the **catch** block that handles the exception. The syntax of the **throw** statement is as follows:

```
throw [expression];
```

Now, let's create a console application named **ThrowStatement** to demonstrate the use of the **throw** statement. Perform the following steps to create the **ThrowStatement** application:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **If Example** application in the **The if Statement** section of this chapter.
2. *Enter* **ThrowStatement** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In this case, we have entered the location as, **D:\C# 2010 Applications.**
3. *Click* the **OK** button. The **ThrowStatement** application is created.
4. *Add* the highlighted code given in Listing 3.12 in the **Program.cs** file:

Listing 3.12: Using the throw Statement

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ThrowStatement
{
   class Program
   {
           static void Main(string[] args)
           {
                   int number;
                   Console.Write("Enter a Number:");
                   number = int.Parse(Console.ReadLine());
                   try
```

```
            {
                if (number > 10)
                throw new Exception("Maximum Limit is 10");
            }
            catch (Exception e)
            {
                Console.WriteLine("An Exception has been Occured");
            }
            finally
            {
                Console.WriteLine("This is the Last Statement");
                Console.Write("\nPress ENTER to quit...");
                Console.ReadLine();
            }
        }
    }
}
```

In Listing 3.12, an integer variable, **number,** is defined. It is used to accept a number as input from a user. A **try** block is defined that encloses an **if** statement. The condition inside the **if** statement checks if the value of the **number** variable is greater than **10**. If this condition evaluates to **true**, then an exception is thrown using the **throw** keyword. The exception thrown in the **try** block is identified in the **catch** block. The **Console.WriteLine()** statement inside the **catch** block is executed and the program control transfers to the **finally** block. The **Console.WriteLine()** statements inside the **finally** block are then executed.

5 *Press* the **F5** key to run the **ThrowStatement** application. The output appears, as shown in Fig.C#-3.19:

Fig.C#-3.19

As seen in Fig.C#-3.19, you are prompted to enter a number.

6 *Enter* any number and *press* the **Enter** key. The output now appears, as shown in Fig.C#-3.20:

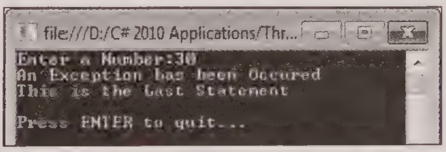


Fig.C#-3.20

Now that you have learned how to handle exceptions in C#, let's move on to learn how to comment a C# program in the following section.

Commenting a C# Program

Comments play an important role in any programming language. Comments are a brief but explanatory note added to the code for the benefit of the readers. You can add comments to the code to enhance the readability of the code by describing the functional characteristics of the code. Any commented code does not pass to the compiler for processing.

In C#, you can add the following two types of comments to your code:

- Single-line comment
- Multi-line comment

Now let's discuss about these in detail.

Single-Line Comments

Single-line comments are those comments that are preceded by double forward slash and can be inserted anywhere in the program as per your desire. The following code snippet shows an example of a single-line comment:

```
string str;         //a string variable is declared
```

When you run the code, the complier ignores the text written after the // symbol.

Multi-Line Comments

There may be some situations where your comments might span across multiple lines. Though you can put a double forward slash and start writing the comments on each line; however, this way your code will end up looking untidy. Therefore, you can use multi-line comments when you want the compiler to ignore multiple lines in a single instance. Similar to the single-line comments, multi-line comments can also be inserted anywhere in a C# program.

The following code snippet shows an instance of a multi-line comment:

```
/*   This is an example of using comments
        in C# code. You can use comments
        for better understandability.
*/
```

In a multi-line comment, the complier ignores the text written between the symbols /* and */.

Note

You can comment an entire block of code by using the Ctrl+K, Ctrl+C keys in combination. To uncomment a selected piece of text, you can press the Ctrl+K or Ctrl+U keys in combination.

With this, we come to the end of the chapter. Let's now summarize the main topics of the chapter.

Summary

In this chapter, you have learned about:

- Different types of statements used in C#
- Selection statements that change the flow of the program based on certain conditions
- Iteration statements or loops that help to execute a C# code for a given number of times
- Jump statements that allow to move the program control from one point to another at any particular instance during the execution of the program
- Exceptions, their types, and how to handle them
- Different types of comments

Chapter 4

Introducing Object-Oriented Programming Constructs

In this Chapter:

- Working with C# 2010 Classes and Objects
- Creating a Structure
- Working with Properties
- Introducing Indexers
- Implementing Encapsulation
- Implementing Inheritance
- Implementing Polymorphism
- Working with Interfaces
- Working with Namespaces

Computer programming, in simple words, means giving instructions to a computer to process the data and provides the required output for performing a particular task. There are mainly two types of programming approaches: procedure-oriented programming (POP) and object-oriented programming (OOP).

POP is a programming methodology wherein a particular problem is divided into smaller sections of code containing a finite number of steps. Each of these sections of code is then solved separately. The steps required to solve a problem and produce the desired outcome are first determined, and then integrated into a single place called a procedure. A procedure call is then used to invoke a procedure. In addition, a procedure can call another procedure, which means that the procedures that you define in an application to solve a problem are dependent on each other. This dependence of a procedure on another procedure makes reusability of an application difficult. Some programming languages that use the POP concept are COBOL, FORTRAN, and C.

The concept of OOP revolves entirely around objects and their interaction to design applications and computer programs, which you learn later in the chapter. An object is an atomic entity having its own identity, state, and behavior. The concept of OOP has been introduced to overcome the difficulty of limited or no reusability of code in POP languages. It also adopts the approach of combining the independent objects, which interact with each other to solve the overall problem and execute the business logic of the program. Between these two approaches, that is, POP and OOP, OOP is considered as the better one, since it follows a model related to any real-world object, which can be anything from a book to a library. It also ensures reusability of the methods, created in one application, in other applications. Examples of languages that follow OOP concepts are C++, JAVA, and C#.

In this chapter, you learn about the four main principles of OOP: encapsulation, inheritance, abstraction, and polymorphism. You also learn about constructors, destructors, and the garbage collection mechanism in C#. In addition, you learn about classes and objects, structures, properties, indexers, interfaces, and namespaces used in C#. Finally, you learn about the concepts of OOP in context with the C# programming language.

Working with C# 2010 Classes and Objects

A class is the primary building block of a program created in any programming language following the OOP concept, such as C#. You can use various classes to encapsulate variables and methods into a single unit. Let's consider an example to understand the concept of classes better. Suppose you need to create an object of a class named **Bird** in your program. To do so, you first need to create a class called **Bird**, which contains all the functionalities or behaviors and properties of any bird. You can then use the **Bird** class to create the objects of the **Bird** class, as needed. Further, you can use the **Bird** class as a template to create an object named **Owl**. The **Owl** object of the **Bird** class would contain a property nocturnal, which would imply that owl is a nocturnal bird.

Classes allow you to define a self-contained environment, wherein you control all the methods that can be applied to a given set of data and also control access to the data. Classes also give you the ability to extend a runtime environment and reuse the existing functionalities of the runtime environment in a new way.

You can create a class with the **class** keyword, followed by the class name. A class can be thought of as a container that may have data members (variables, constants, or fields) and member functions (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and other classes. A class also supports inheritance, which is a mechanism in which a derived class extends a base class. The syntax of class in C# is:

```
class <className>
{
      <access-modifier> [static] <return-type> MethodName( <signature>);
      <access-modifier> [static] <variable type> variablename;
}
```

Note

You learn about inheritance later in the chapter.

To access the data members and member functions of a class, you need to declare an object of that class. The syntax to declare an object in C# is:

```
<className> <ObjectName>= new <className>();
```

C# provides many ways of defining classes, such as providing different access levels, inheriting features from other classes, and enabling you to specify and control the behavior and output of the code when objects are instantiated or destroyed. To implement all these in your C# application, you need to know about the following concepts:

- Access modifiers
- Methods
- Constructors and destructors
- Partial classes
- Static classes
- Extension methods

Let's learn about each of these in detail in the following sections.

Introducing Access Modifiers

A class member or type's accessibility level can be specified using access modifiers. An access modifier helps avoid jumbling of data and methods with the existing code, as well as protects an object of a class from outside interference. It protects an object by defining a certain scope to access data and methods in a restricted manner. You can declare a class and its methods with an access modifier. However, one method can contain only one modifier. The different types of access modifiers used in C# are listed in Table 4.1:

Table 4.1: Access Modifiers in C#

Access Modifiers	Description	Syntax

Public	Gives public access to members both inside and outside a class without any restrictions.	`class SampleClass`
		`{`
		`    public int x; // No access`
		`                 // restrictions.`
		`}`

Protected	Gives protected access to members. You can access protected members from either the class in which they are declared or a class derived from the class in which they are declared.	`class A`
		`{`
		`    protected int x = 123;`
		`}`

Internal	Gives access to the members that are in the current assembly. If a member with internal access modifier is accessed outside the assembly in which it has been defined, an error is generated in an application.	`public class BaseClass`
		`{`
		`    // Only accessible within the`
		`    // same assembly`
		`    internal static int x = 0;`
		`}`

Private	Gives access to the members that are within the body of the class in which they are declared.	`class Employee`
		`{`
		`    private int i;`
		`    double d;   // private access`
		`    by default`
		`}`

Table 4.1: Access Modifiers in C#

Access Modifiers	Description	Syntax
Protected internal	Gives access to the members that are visible either to the current assembly or to the types derived from the class in which they are declared.	`namespace MyAssembly//current //assembly { public class One { protected internal int Method() { //method code } } public class Two : One { public int Add() { One obj = new One(); return obj.Method(); } } }`

Note

An assembly is a collection of one or more files that are used as a single unit. It forms a single unit of deployment and is the primary building block of a .NET application.

Next, let's discuss about methods and how you can work with them in C#.

Working with Methods

A method in C# is a block of code that contains a series of statements to perform an action. Every action in C# is performed in the context of a method. Methods can be declared inside a class or a struct.

Note

A struct is a value type that consists of constructors, constants, data members, methods, and more. You can create a structure in C# by using the struct keyword. You learn more about structure in C# in detail in the later chapter.

A method declaration typically includes access modifiers that specify the access level, the return value, the name of the method, and the method parameters. All these combined together is called the signature of the method.

The following code snippet shows how you can declare a method:

```
class Passenger
{
        public void Title() { }
        public void Name(int age) { }
        public int Address(int houseno, int pincode)
        {
                //Some code
        }
}
```

The code in the preceding snippet contains three methods, namely, **Title()**, **Name()**, and **Address()**. The parameters of these methods are enclosed in parenthesis and separated by comma. If you pass a set of empty parenthesis, it implies that the method requires no parameters.

Constructors and destructors are special methods of every class in a C# program. Each class has its own constructor and destructor, which are called automatically when the instance of a class is created or destroyed. The constructor initializes all class members whenever you access a class, and the destructor destroys the values assigned to the class members when the objects are not required anymore.

Note

You learn about constructors and destructors in detail later in the chapter.

Now, let's learn how you can define and then call a method in a C# program.

Defining a Method

As discussed earlier, a method consists of a sequence of statements. These statements constitute the method body and are executed when the method is called. A method, in general, defines the behavior of the class and allows the programmer to separate the program logic into different units. You can always pass some information to a method, execute an instruction or a set of instructions, and then have the value returned from it.

Let's consider a simple example to understand the concept of methods better. Suppose you have to find the square of the number 10. You may perhaps write the code as shown in the following code snippet:

```
int square = 10 * 10;
```

Now consider a situation where you need to accept input from the user. The preceding code snippet will not serve the purpose anymore. However, using a method, you can perform this task quite easily.

Let's perform the following steps to create a console application named **CreatingMethod** and learn how to define a method in a C# program:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 Integrated Development Environment (IDE).
2. In the Visual Studio 2010 IDE, *select* the **File→New→Project** option from the menu bar to open the **New Project** dialog box.
3. In the **New Project** dialog box, *select* the **Visual C#** option in the **Installed Templates** pane and the **Console Application** option in the middle pane.

   Enter **CreatingMethod** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In our case, we have selected the default location, which is **D:\C# 2010 Applications**.

5. *Click* the **OK** button. The **CreatingMethod** application is created.
6. In the **Program.cs** file, *add* the highlighted code snippet given in Listing 4.1 inside the **Program** class:

Listing 4.1: Showing the code for Defining a Method

```
public static int GetSquare()
{
        int num, square = 0;
        Console.WriteLine("Enter an integer number");
        num = Convert.ToInt32(Console.ReadLine());
        square = num * num;
        return square;
}
```

In Listing 4.1, a method, **GetSquare()** is defined. Two local integer variables, namely, **num** and **square**, are declared and initialized with value **0**. The **Console.WriteLine()** statement is used to prompt to the user to enter

an integer number. The variable **square** is assigned the square of the number entered by the user. The value contained inside the variable **square** is then returned.

Calling a Method

In C#, after defining a method in your code, you need to call it to implement the code and execute the logic of the program. Calling a method is similar to accessing a file from a different location on a computer. You can call a method in the **Main()** method of the **Program** class of an application.

Let's now learn how you can call a method after defining it. We shall call the method defined in Listing 4.1.

Perform the following steps to call the **GetSquare()** method defined in the **CreatingMethod** application created in the **Defining a Method** section of this chapter:

1. *Add* the highlighted code snippet given in Listing 4.2 to the **Main()** method of the **Program** class in the **CreatingMethod** application:

Listing 4.2: Calling a Method

```
static void Main(string[] args)
{
      Console.WriteLine("The square of the number is " + Program.GetSquare());
      Console.WriteLine("\nPress ENTER to quit..");
      Console.ReadLine();
}
```

2. *Press* the **F5** key to run the **CreatingMethod** application.

   The output of the application appears, as shown in Fig.C#-4.1:

Fig.C#-4.1

3. *Enter* a number and *press* the **Enter** key on the keyboard. The output appears, as shown in Fig.C#-4.2:

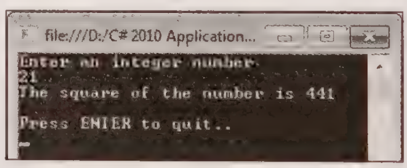


Fig.C#-4.2

As shown in Fig.C#-4.2, when you enter the number **21**, its square is calculated and displayed on the screen. Now, let's examine how to pass parameters to a method in C#.

Passing Parameters

In C#, using the **params** keyword in your C# application, you can define a method parameter. You can use the **params** keyword to accept variable number of arguments. The **params** keyword takes the last position in the method's list of formal arguments. To create a method that accepts variable number of arguments, you can perform the following steps:

1. *Repeat* steps 1 to 3 of the **Defining a Method** section of this chapter.
2. *Enter* **PassingParameters** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In our case, we have selected the default location, which is **c:\users\temp\documents\visual studio 2010\Projects.**
3. *Click* the **OK** button. The **PassingParameters** application is created.

4. In the **Program.cs** file, *add* the code snippet given in Listing 4.3:

Listing 4.3: Passing Parameters to a Method

```
public static int GetSum(params int[] numbers)
{
        int sum = 0;
        foreach (int num in numbers)
                sum += num;
        return sum;
}
```

In Listing 4.3, a method called **GetSum()** is defined, which takes variable number of arguments. An integer variable, sum is declared and initialized to 0 inside the **GetSum()** method. The **foreach** loop is used to iterate over all the numbers the **GetSum()** method accepts as arguments and assign the sum of all these numbers to the variable sum. Finally, the value inside the variable sum is returned.

To call the **GetSum()** method, perform the following steps:

1. *Call* the **GetSum()** method by *adding* the highlighted code snippet given in Listing 4.4 to the **Main()** method of the **Program** class in the **PassingParameters** application:

Listing 4.4: Calling a Parameterized Method

```
static void Main(string[] args)
{
        Console.WriteLine("The Sum is:");
        Console.WriteLine(GetSum(1, 2, 3, 4, 5, 6, 7, 8, 9));
        Console.WriteLine(GetSum(2, 4, 6, 8));
        Console.WriteLine("\nPress ENTER to quit..");
        Console.ReadLine();
}
```

In Listing 4.4, the **GetSum()** method is first called by passing nine numbers as arguments and then called again by passing four numbers as arguments.

2. *Press* the **F5** key to run the **PassingParameters** application.

The output of the application appears, as shown in Fig.C#-4.3:

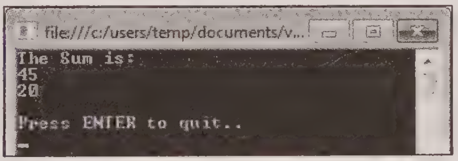


Fig.C#-4.3

In Fig.C#-4.3, you can see that when the **GetSum()** method is called first, it returns the sum of all the nine numbers; when the **GetSum()** method is called by passing four numbers as arguments, it returns the sum of all the four numbers.

Now that you know how to pass parameters to a method, let's discuss the ways in which you can pass parameters to a method. C# allows methods to be passed either by value or by reference. Let's first discuss how to pass a parameter by value in the following section.

Pass by Value

When a parameter is passed by value, a copy of the variable is passed into the method being called. The changes to the parameter that take place inside the method body do not affect the original data. Let's perform the following steps to pass a parameter by value:

1. *Repeat* steps 1 to 3 of the **Defining a Method** section of this chapter.

2. *Enter* **PassByVal** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In our case, we have selected the default location, which is **c:\users\temp\documents\visual studio 2010\Projects.**
3. *Click* the **OK** button. The **PassByVal** application is created.
4. In the **Program.cs** file, *add* the highlighted code snippet given in Listing 4.5:

Listing 4.5: Passing Parameters by Value to a Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PassByVal
{
    class Program
    {
        public static void Square(int i)
        {
            i = i*i;
            Console.WriteLine("The value inside the method: {0}", i);
        }
        static void Main(string[] args)
        {
            int num = 15;
            Console.WriteLine("The value before calling the method: {0}",
               num);
            Square(num);
            Console.WriteLine("The value after calling the method: {0}",
               num);
            Console.ReadLine();
        }
    }
}
```

In Listing 4.5, the method **Square()** is defined, which takes an integer variable **i** as argument. The **Square()** method simply displays the value contained inside **i** on the screen. The **Square()** method is called inside the **Main()** method. A local variable **num** is declared and assigned the value **15** inside the **Main()** method. Since the variable **num** has been passed by value, any changes made to it will not have any effect on its original value.

5. *Press* the **F5** key to run the **PassByVal** application.

   The output of the application appears, as shown in Fig.C#-4.4:

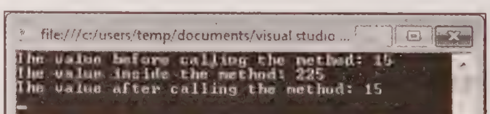


Fig.C#-4.4

In Fig.C#-4.4, you can see that initially before any call to the method **Square()** has been made, the value of **num** is **15**. When the method is called, the value of **num** is copied into the local variable **i** and its square is computed. However, the value of **num** does not change inside the **Main()** method and the value **15** is returned. This is due to the fact that a change in the value only occurs inside the **Square()** method and affects the variable **i**.

Pass by Reference

When you want to treat the C# value types as reference types or simply wish to retain any changes made to the value of a parameter in a method, you can use the pass by reference mechanism. In other words, it means that when you pass a value by reference, you actually pass the address of that value and not a copy. Therefore, any changes made to it inside the method persist outside the method as well. In C#, a parameter is passed by reference by using the **ref** and **out** keywords. We shall, however, describe how to pass a parameter by reference using the **ref** keyword only. Let's perform the following steps to pass a parameter by reference:

1. *Repeat* steps 1 to 3 of the **Defining a Method** section of this chapter.
2. *Enter* **PassByRef** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In our case, we have selected the default location, which is **C:\Users\temp\documents\visual studio 2010\Projects.**
3. *Click* the **OK** button. The **PassByRef** application is created.
4. In the **Program.cs** file, *add* the highlighted code snippet given in Listing 4.6:

Listing 4.6: Passing Parameters by Reference to a Method

```
    class Program
{
    public static void ChangeValue(ref int x)
    {
        x = x * x;
        Console.WriteLine("The value inside the method: {0}", x);
    }
    static void Main(string[] args)
    {
        int num = 5;
        Console.WriteLine("The value before calling the method: {0}", num);
        ChangeValue(ref num);
        Console.WriteLine("The value after calling the method: {0}", num);
        Console.ReadLine();
    }
}
```

In Listing 4.6, a method called **ChangeValue()** is defined, which takes the value of an integer variable **x** as argument. The argument value is passed using the **ref** keyword in the call for the **ChangeValue()** method. The **ChangeValue()** method computes the square of the value inside the variable **x** and displays the same on the screen. A local integer variable **num** is declared and assigned the value **5** inside the **Main()** method. A call to the **ChangeValue()** method is made by passing the value of variable **num** by reference; therefore, any changes made to **num** will cause its value to be changed outside the **ChangeValue()**method.

5. *Press* the **F5** key to run the **PassByRef** application.

The output of the application appears, as shown in Fig.C#-4.5:

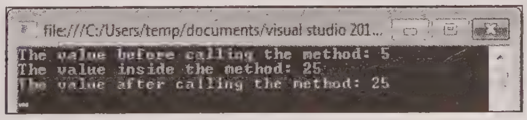


Fig.C#-4.5

In Fig.C#-4.5, you can see that initially before any call to the method **ChangeValue()** has been made, the value of num is **5**. During the call to the method, the argument's reference is passed, due to which the method accesses the referred memory location and performs the necessary modifications on the variable at its memory location. Therefore, the original value of the variable num is changed after the call to the method **ChangeValue()**.

Working with Constructors and Destructors

Constructors are special methods used to instantiate a class or when an object is first created in an application. You can use a constructor to initialize objects and set any parameters. In addition, you can write constructors to accept arguments. The constructor is called by the same name as the class, while the name of the destructor is made up of the class name prefixed with a tilde symbol (~).

The syntax to define a constructor is as follows:

```
public class-name (parameter-list)
{
        //body
}
```

The main features of constructors are as follows:

- Have the same name as the class itself
- Do not have any return type
- It is not mandatory to declare a constructor; it is invoked automatically

A destructor (or finalizer) is called when an object is finally destroyed. Destructors are used to destruct the instances of classes when the instances are not required. A destructor is always invoked automatically, so you cannot call it in your application. C# provides a garbage collection mechanism that is executed when the runtime environment finds it necessary or when an object is not destroyed until its reference count drops to 0. You have no means to inform when an object will be destroyed and when the destructor will be called. You can, however, implement a custom method that allows you to control object destruction by calling the destructor.

Let's perform the following steps to create an application named **ConstructorApp** to learn how to use constructors:

1. *Repeat* steps 1 to 3 of the **Defining a Method** section of this chapter.
2. *Enter* **ConstructorApp** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **ConstructorApp** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.7:

Listing 4.7: Defining a Constructor

```
class Program
{
        int length, breadth;
        public Program(int x, int y)
        {
                length = x;
                breadth = y;
        }
        public void area()
        {
                int result = length * breadth;
                Console.WriteLine("Area is : " + result);
        }
        static void Main(string[] args)
        {
                Program prg = new Program(10, 20);
                prg.area();
                Console.ReadLine();
        }
}
```

In Listing 4.7, we have created one constructor named **Program** (you already know that the constructor has the same name as the class). In the **Main()** method, we have initialized the constructor and passed the parameters required to initialize the values of different variables. Then, we have accessed the **area()** method to calculate the area of the rectangle.

5. *Press* the **F5** key to run the **ConstructorApp** application.

   The output of the application appears, as shown in Fig.C#-4.6:

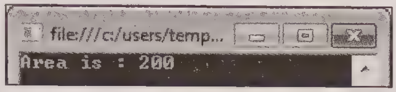


Fig.C#-4.6

Next, let's discuss about the static constructors.

Static Constructors

A static constructor is the one that can be used to access the static members of a class and therefore, a static contsructor is often used to initialize the static members of a class. You can also use a static constructor when you need to perform any particular task that has to be performed just once. A static constructor is automatically invoked for the first time a class loads into the memory and there can be only one static constructor in a class. The syntax to define a static constructor is as follows:

```
public class StaticConstructorClass
{
        static StaticConstructorClass()
   {
    // some code
   }
}
```

The preceding code snippet shows that the name of a static constructor is not prefixed by an access modifier; instead, it is preceded by the keyword **static**. The following are some of the main properties of a static constructor:

- A static constructor's name is neither prefixed by any access modifier nor does it takes any arguments
- It is automatically invoked for the first time to initialize a class even before any of its instances has been created or any other static data member is referenced
- It cannot be called directly
- A user has no control over the execution of a static constructor in a program

Let's perform the following steps to create an application named **StaticConstructor** to see how a static constructor is created and invoked:

1. *Repeat* steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **StaticConstructor** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **StaticConstructor** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.8:

Listing 4.8: Creating a Static Constructor

```
namespace StaticConstructor
{
   public class StaticTest
    {
      static StaticTest()
```

```
            {
                Console.WriteLine("Executing the Static Constructor");
            }
            public StaticTest()
            {
                Console.WriteLine("Executing a non-static Constructor");
            }
        }
        class Program
        {
            static void Main(string[] args)
            {
                StaticTest st = new StaticTest();
                Console.ReadLine();
            }
        }
    }
```

In Listing 4.8, we have created a static constructor and a non-static constructor. Inside the **Main()** method, the non-static constructor of the class **StaticTest** is invoked automatically when an object of that class is created using the **new** operator. Note that we have not made any call to the static constructor since a static constructor is automatically invoked the first time a class loads into the memory.

5 *Press* the **F5** key to run the **StaticConstructor** application.

The output of the application appears, as shown in Fig.C#-4.7:

Fig.C#-4.7

In Fig.C#-4.7, you can see that when the class **StaticTest** is loaded for the first time in the memory, the static constructor executes first, irrespective of the fact that the class already has a non-static constructor defined. After the static constructor is executed, the non-static constructor of the class **StaticTest** executes.

Instance Constructors

The constructors that are used to create and initialize the instance variables of a class using the new operator are known as instance constructors. The following code snippet illustrates how to define an instance constructor:

```
public class Numbers
{
      int a,b;
  public Numbers() // instance constructor
 {
  a = 0;
  b = 0;
 }
}
```

In the preceding code snippet, an instance constructor is defined. It is invoked whenever an object of the class **Numbers** is created. The constructor in the preceding code snippet takes no arguments; however, we can add such constructors to our code that take some arguments. Such constructors are known as parameterized constructors. For example, we can add a parameterized constructor to the class **Numbers**, as shown in the following code snippet:

```
public Numbers(int a, int b) // parameterized constructor
{
```

```
        this.a = a;
        this.b = b;
     }
   }
```

The instances of a class can be created either by invoking the instance constructor or by calling the parameterized constructor or both, as shown in the following code snippet:

```
Numbers num1 = new Numbers(); //default constructor
Numbers num2 = new Numbers(1, 2); //parameterized constructor
```

When no constructor for a class has been defined explicitly by the programmer in the source code, the default constructor for that class is automatically called and the value of the instance variables is initialized to their default values. Now, let's perform the following steps to create an application named **ConstructorTypes** to understand how the default and parameterized constructors work:

1. *Repeat* steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **ConstructorTypes** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **ConstructorTypes** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.9:

Listing 4.9: Demonstrating an Instance and Parameterized Constructor

```
namespace ConstructorTypes
{
    public class Numbers
    {
        int a, b;
        public Numbers() // instance constructor
        {
            a = 0;
            b = 0;
        }
        public Numbers(int a, int b) //parameterized constructor
        {
            this.a = a;
            this.b = b;
        }
        public void ShowNumbers()
        {
            Console.WriteLine("The numbers are {0} and {1}", a,b);
        }
    }
   public class Program
    {
       public static void Main(string[] args)
        {
            Numbers num1 = new Numbers();
            Numbers num2 = new Numbers(2, 3);
            num1.ShowNumbers();
            num2.ShowNumbers();
            Console.ReadLine();
        }
    }
}
```

In Listing 4.9, two integer instance variables, **a** and **b**, are declared. An instance constructor is defined as that does not take any parameters. A parameterized constructor is also defined as that takes two

parameters. In the **Main()** method, we have invoked both the constructors. Thereafter, the **ShowNumbers()** method is invoked to display the values of instance variables, **a** and **b,** in the objects, **num1** and **num2,** of the class **Numbers**.

5 *Press* the **F5** key to run the **ConstructorTypes** application.

The output of the application appears, as shown in Fig.C#-4.8:

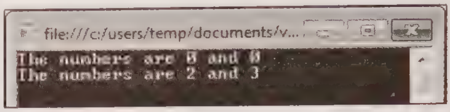


Fig.C#-4.8

In Fig.C#-4.8, when the instance constructor is called, the instance variables **a** and **b** are initialized to **0**; on the other hand, on invoking the parameterized constructor, the values of the instance variables are initialized as per the arguments passed to the constructor.

In this section, you have learned how to create objects using constructors. However, there may be times when you need to release these objects. The next section discusses how you can free unused objects.

Garbage Collection in C# 2010

The garbage collection feature of .NET Framework is responsible for the management of resources such as memory. It is due to this memory management aspect of .NET Framework that a .NET application developer need not worry about allocating and then releasing memory for objects in an application. In other words, it can be said that with the help of this feature, the .NET developers do not require to write code to perform memory management tasks. Therefore, the automatic garbage collection mechanism in C# helps prevent issues, such as memory leaks caused by forgetting to free an unused object, null pointer exceptions that are caused when an attempt is made to access the memory of an object that has long been freed, and many more.

In C#, the garbage collection mechanism is managed by common language runtime (CLR). As long as there is free space in the managed heap, new objects are automatically allocated memory by CLR. Therefore, the garbage collection is performed periodically by the Garbage Collector (GC) to reclaim the memory from any unreferenced objects and to make it available for subsequent allocation to some other object. The garbage collection mechanism also enforces memory safety, which means that the contents of one object cannot be accessed by some other object.

As stated earlier, garbage collection is performed periodically for .NET applications; yet there might be times when you need to force the GC to deallocate memory from any unused object. For this purpose, you can call the **GC.Collect()** method. This method causes an immediate garbage collection on objects of all generations. The synatx to call the **GC.Collect()** method to reclaim inaccessible memory from objects of generation 0 is as follows:

```
GC.Collect(0);
```

In .NET Framework 4.0, support for background garbage collection has been added as opposed to the concurrent garbage collection that was supported in the earlier versions of .NET Framework. This background garbage collection feature means that garbage collection can be simultaneously performed on generation 0 and generation 1 objects while the garbage collection is being performed on generation 2 objects.

Note

The .NET GC is a generational garbage collector. In fact, the managed heap has been partitioned into three generations, which are known as generation 0, generation 1, and generation 2. A newly created object is stored in the generation 0 by the GC. As the memory is finite and there may soon come a time when there is no free space available in the managed heap: therefore, the GC will deallocate objects stored in generation 0. Those objects that survive the first pass of garbage collection are then moved on to generation 1. The generation 2 consists of those objects that survive the first two garbage collection passes. The objects stored in generation 2 are also known as long-lived objects.

In this section, you learned about the garbage collection mechanism and how you can enforce garbage collection for any unused resources in your application. Now, let's learn about the **IDisposable** interface in the next section.

The IDisposable Interface

Garbage collection in .NET applications is non-deterministic in nature. This means that one cannot predict when garbage collection will be performed by the GC. Therefore, the need to explicitly enforce garbage collection arises for applications that have significant memory requirements. In such cases, it is recommended to implement the **IDisposable** interface. This interface defines a method to free allocated unmanaged resources because the GC does not possesses any knowledge about unmanaged resources, such as open files and streams. The **IDisposable** interface defines the **Dispose()** method that is responsible for the release of managed as well as unmanaged resources. The **Dispose()** method is used in conjunction with the GC.

Note

When you implement the IDisposable interface, use the try ... catch ... finally block to ensure that the unmanaged resources are released despite the occurrence of any exception. An unmanaged resource is anything that is not included within the .NET Framework, such as the operating system resources, which include a text file or a window. Network or database connections are also examples of unmanaged resources.

Next, let's learn about the methods used to release any resources owned by an object of a .NET application.

The Dispose and Close Methods

The **Dispose()** method is defined inside the **IDisposable** interface and is used to release any unmanaged resources. This is the preferred method to release resources that are assigned to an object or to prepare the object for reuse. Let's consider the Listing 4.10, which defines the **Dispose()** method:

Listing 4.10: Defining the Dispose() method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ReleasingObject
{
        class Program : Idisposable
        // class Program implements the Idisposable interface
        {
           // calling the Dispose() method
           public void Dispose()
           {
            // some code
                Console.WriteLine("Outside the Dispose() method!");
           }
   }
```

In Listing 4.10, the **IDisposable** interface is implemented by the **Program** class, which contains the **Dispose()** method. You can add code to perform any clean-up operations inside the **Dispose()** method.

In .NET Framework, there is another method to release resources, which is known as the **Close()** method. The **Close()** method is executed to temporarily close any managed resource, such as a database connection. This implies that a particular resource for which you have called the **Close()** method can be reopened. However, the **Dispose()** method removes an unmanaged resource permanently from the memory and that resource is not available for any further processing.

Working with Partial Classes

A partial class in C# is the class that enables you to specify the definition of a class in two or more source files. All the source files contain a section of the class definition. The definitions in the different source files are

combined when the application is executed. You can distribute a partial class over multiple separate files, so that you can work on the separate files simultaneously by using one partial class.

You can declare a partial class by using the **partial** keyword. The **partial** keyword indicates that all the parts of the class must be available at compile time to generate the final type.

Let's perform the following steps to create an application named **PartialClass** to learn how to use a partial class:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **PartialClass** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box
3. *Click* the **OK** button. The **PartialClass** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.11:

Listing 4.11: Using a Partial Class

```
namespace PartialClass
{
    public partial class Orders
    {
        private int x;
        private int y;
        public Orders(int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    }
    public partial class Orders
    {
        public void PrintOrders()
        {
            Console.WriteLine("Orders: {0}, {1}", x, y);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Orders ord = new Orders(20, 60);
            ord.PrintOrders();
            Console.ReadLine();
        }
    }
}
```

In Listing 4.11, the fields and constructors of the **Orders** class are declared in one partial class definition, and the **PrintOrders()** method is declared in another partial class definition.

5. *Press* the **F5** key to run the **PartialClass** application.

The output of the application appears, as shown in Fig.C#-4.9:

Fig.C#-4.9

Fig.C#-4.9 shows that all partial classes are compiled as a single class and then executed. In the next section, you learn about static classes.

Working with Static Classes

A **static** class in C# is the same as a non-static class; however, unlike a non-static class, it cannot be instantiated. In other words, it means that when you declare a class as **static**, then you cannot use the **new** keyword to create an object of that class. Static classes in C# can be used to create data members and functions, which are accessible even if you do not create any objects of the class. You can access the members of a static class by using the class name itself. The **static** keyword is used to define a class and its members as static.

A static class does not contain instance constructors, which are used to create and initialize instances. The features of a static class are as follows:

- A static class can contain only static members
- A static class cannot be instantiated
- Static classes are sealed
- You cannot inherit static classes

Let's now learn how to declare a static class and use a static method in a class.

Declaring a Static Class

When you declare a static class in your C# application, you must consider the basic rule that static classes cannot be instantiated. Declaring a static class is similar to creating a class that contains only static members and a private constructor. The following code snippet shows how to declare a static class:

```
static class Employee
{
      public static int id;
      public static int tele_no;
}
```

In the preceding code snippet, **Employee** is the static class, and **id** and **tele_no** are its static data members. If a class is static, its members are also static.

Using a Static Method in a Class

You can call a static method within a class, in which the static method is declared, without creating the instance of the class. If you create an instance of the class, you cannot use it to call static members. A static member can exist inside a static or a non-static class.

Let's perform the following steps to create an application named **StaticMethod** to learn how to use a static method in a class:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **StaticMethod** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **StaticMethod** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.12:

Listing 4.12: Showing the Code to Add a Static Method in a Class

```
namespace StaticMethod
{
      static class MathFunction
      {
            static public double Square(double num)
            {
```

```
                return num * num;
            }
            static public double Fraction_Part(double num)
            {
                return num - (int)num;
            }
        }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Square of 5 is " + MathFunction.Square(5.0));
            Console.WriteLine("Fractional   part   of   4.234   is   "   +
        MathFunction.Fraction_Part(4.234));
            Console.ReadLine();
        }
    }
}
```

In Listing 4.12, two static methods, **Square()** and **Fraction_Part()**, are defined inside a static class, namely, **MathFunction**. As the class **MathFunction** is declared as static, therefore the **Square()** and **Fraction_Part()** methods defined inside it are invoked using the class name itself.

5 *Press* the **F5** key to run the **StaticMethod** application. The output of the application appears, as shown in Fig.C#-4.10:

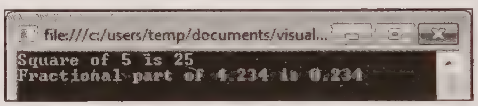


Fig.C#-4.10

Now that you have learned about static classes and static methods, let's discuss about Extension methods in the next section.

Using Extension Methods

Extension method is a feature that was introduced in C# 2008. An extension method is a specific type of method used to extend a class without inheriting the class. Similar to a static method, an extension method is also defined in a static class. The first parameter of an extension method is the **this** modifier, which specifies the type, the extension method is going to extend. The differences between an extension method and static method are listed in Table 4.2:

Table 4.2: Difference between Static Method and Extension Method

Extension Method	Static Method
An extension method uses the **this** keyword as its first method parameter	A static method does not use the this keyword before its first argument
When an extension method is called, the argument declared with the **this** keyword is not passed	Passing all the arguments is necessary to call a static method
An extension method is defined in a static class only	A static method can be defined in all classes

Let's perform the following steps to create an application named **ExtensionMethod** to learn how to use an extension method:

1 *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.

2. *Enter* **ExtensionMethod** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **ExtensionMethod** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.13:

Listing 4.13: Implementing an Extension Method

```
namespace ExtensionMethod
{
      public static class MyClass
      {
            public static int MyExtensionMethod(this string Number)
            {
                  return Int32.Parse(Number);
            }
      }
      class Program
      {
            static void Main(string[] args)
            {
                  string Number = "100";
                  int ext = Number.MyExtensionMethod();
                  Console.WriteLine("The Output from MyExtensionMethod()");
                  Console.WriteLine(ext);
                  Console.Write("\nPress ENTER to quit...");
                  Console.ReadLine();
            }
      }
}
```

In Listing 4.13, an extension method, **MyExtensionMethod()**, is defined. The **MyExtensionMethod()** method has the **this** keyword before its string type argument, **Number**.

5. *Press* the **F5** key to run the **ExtensionMethod** application.

The output of the application appears, as shown in Fig.C#-4.11:

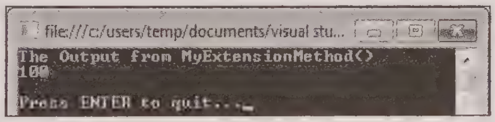


Fig.C#-4.11

This completes our discussion on how to define a class in C#. Now, let's move on to know how to create a structure in the next section.

Creating a Structure

A structure in C# is a user-defined value type, which is similar to the built-in type objects such as int, float, bool, and so on. A structure declaration is analogous to that of a class and may consist of constructors, constants, fields, methods, properties, indexers, operators, and nested types. The **struct** statement creates a structure of elements in C#. The accessibility level of all elements of a structure is **private** by default, which means that the elements of a structure have private scope. In addition, a structure initializes its elements to the default value for each data type if none is specified. In C#, one significant difference between structures and classes is that a structure does not support inheritance. If you do not use the new operator to call a constructor when you are declaring a structure variable, the structure object is created, but the values of the structure variable are unassigned.

The syntax of a **struct** statement is similar to that of a class, with the main difference being that a **struct** is a value type and a **class** is a reference type. The syntax of a **struct** statement is:

```
[attributes] [modifiers] struct identifier [:interfaces]
{
      struct body[;]
}
```

In the preceding syntax, the following are the options used to create a struct:

- **attributes**: Refers to an optional value that can be used to specify additional declarative information.
- **modifiers:** Allows modifiers, such as new, static, virtual, abstract, and override, to be used in a **struct**. It also allows access modifiers, such as public, internal, and private, to be used in a **struct**. This is also optional.
- **struct identifier:** Creates a **struct**. It is a mandatory keyword. An identifier that represents the name of the **struct** must follow the **struct** keyword.
- **interface**: Contains the interface list, separated by commas. This is also optional.
- **struct body**: Contains all member declarations.

Now let's create an application named **Structure** to learn how to create a structure by performing the following steps:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* the name **Structure** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **Structure** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.14:

Listing 4.14: Creating a Structure

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Structure
{
      public struct X
      {
            public int x;
            public int y;
            public int z;
      }
      class Program
      {
            static void Main(string[] args)
            {
                  X x;
                  x.x = 1;
                  x.y = 2;
                  x.z = 3;
                  Console.WriteLine("X = {0}, Y = {1}, Z = {2}", x.x, x.y, x.z);
                  X x1 = new X();
                  Console.WriteLine("X = {0}, Y = {1}, Z = {2}", x1.x, x1.y,
x1.z);
                  Console.Write("\nPress Enter to quit...");
                  Console.ReadLine();
```

```
            }
        }
    }
```

In Listing 4.14, you can see that a structure **X** is declared using the **struct** keyword. The structure X is then called in the **Main()** method to display the value of its elements.

5. *Press* the **F5** key to run the **Structure** application.

   The output of the application appears, as shown in Fig.C#-4.12:

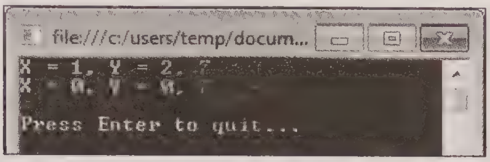


Fig.C#-4.12

Now that you have learned how to create structures, let's move further and learn about the properties in C# in the next section.

Working with Properties

In C#, properties are a standard part of the language itself. A property is a way for you to expose an internal data element of a class in a simple and intuitive manner. C# is one of the first languages to offer direct support for properties. Programming languages, such as C++ Builder and Delphi, offered this feature as an extension to the languages they supported, but these were not considered standard features.

You can implement properties in C# with the type-safe **get()** and **set()** methods. You can create a property by defining an externally available name, and then writing the **set()** and **get()** methods to implement the property. The **get()** method is used to return the property value, and the **set()** method is used to assign a new value.

In this section, you learn about using a property and using an anonymous type for read-only properties.

Using a Property

You can read, write, or compute the value of a private data member of a class in C# using its properties. Using the properties, you can prevent data from external usage and modifications. You can declare a property in which you can use the **get** and **set** accessors to retrieve the required value, and then assign the value to specified data.

Let's perform the following steps to create an application named **Properties** to learn to use properties:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter**.**
2. *Enter* **Properties** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **Properties** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.15:

Listing 4.15: Using a Property

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Properties
{
    public class BookDetail
```

```
    {
        private string Name;
        public string getName()
        {
            return Name;
        }
        public void setName(string name)
        {
            Name = name;
        }
    }
    public class MainClass
    {
        static void Main(string[] args)
        {
            BookDetail bdetail = new BookDetail();
            bdetail.setName("C# 2010 in Simple Steps");
            Console.WriteLine("The Book Name is :" + bdetail.getName());
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
}
```

In Listing 4.15, a private string variable **Name** is declared, and two properties, namely, **getName** and **setName,** are also defined. The **getName** property is used to return the property value from the string variable **Name,** and the **setName** property is used to assign a new property value to the variable **Name**. In the **MainClass** class, you can see that an object of the **BookDetail** class, named **bdetail**, is created. The **set** accessor of the property **setName** is called in the following statement of the preceding list:

```
bdetail.setName("C# 2010 in Simple Steps");
```

The preceding statement in Listing 4.15 assigns the string value **C# 2010 in Simple Steps** to the variable **Name** inside the **setName** property. Similarly, the following statement in the preceding listing calls the **get** accessor of the **getName** property.

```
Console.WriteLine("The Book Name is :" + bdetail.getName());
```

5. *Press* the **F5** key to run the **Properties** application.

The output of the application appears, as shown in Fig.C#-4.13:

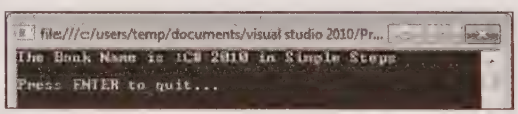


Fig.C#-4.13

This completes our discussion on properties. Next, let's discuss about the anonymous types in C#.

Using an Anonymous Type for Read-Only Properties

While creating properties for your application, you might need to create a few read-only properties. You can encapsulate these read-only properties into a single unit through anonymous types. Anonymous types provide a way to encapsulate the read-only properties of an object without having to first explicitly define a type. This means that while declaring the read-only properties you need not specify its type. The compiler generates the type name at compile time as required, but this type name is not available at the source code level. In other words, anonymous types create unnamed structure types that you can add to the collections and access those structure types using the **var** keyword. The anonymous type declaration starts with the **new** keyword. An

anonymous type provides you an easy way to encapsulate a set of read-only properties into a single object without defining a new type.

Let's perform the following steps to create an application named **AnonymousType** to learn to use an anonymous type for read-only properties:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **AnonymousType** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **AnonymousType** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.16:

Listing 4.16: Using Anonymous Type for Read-Only Properties

```
namespace AnonymousType
{
      class Program
      {
              static void Main(string[] args)
              {
                      var Language = new { Name = "C#", Version = 4 };
                      Console.WriteLine("Language Details");
                      Console.WriteLine("Name:" + Language.Name);
                      Console.WriteLine("Version " + Language.Version);
                      Console.Write("Press ENTER to quit...");
                      Console.ReadLine();
              }
      }
}
```

In Listing 4.16, anonymous types are created that consist of two properties - **Name** and **Version**. These anonymous types are assigned to a variable **Language** that is defined by using the **var** keyword. The two properties are later accessed using the **Language** variable to display the property values.

5. *Press* the **F5** key to run the **AnonymousType** application.

The output of the application appears, as shown in Fig.C#-4.14:

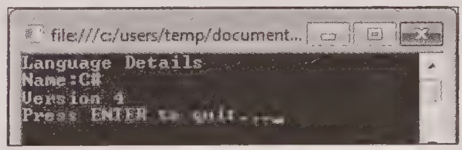


Fig.C#-4.14

The anonymous types are also called **class types** and consist of one or more public read-only properties. However, they do not contain any other class members, such as methods and events. You can use the **new** operator with the object initializer to create an anonymous type. The following code snippet shows an anonymous type, **id** being initialized with two properties called **Age** and **Designation**:

```
var id = new { Age = 30, Designation = "Analyst"};
```

Whenever an anonymous type is initialized, as shown in the preceding code snippet, the compiler converts the anonymous type into a declaration, as shown in the following code block:

```
class _Anonymous1
{
 private int Age;
 private string Designation;
```

```
        public int age
        {
         get { return Age; }
         set { Age = value;}
        }
        public string designation
        {
         get { return Designation; }
         set { Designation = value; }
        }
       }
       _Anonymous1 id = new _Anonymous1();
       id.age = 30;
       id.designation = "Analyst";
```

Now, let's discuss about Indexers in the next section.

Introducing Indexers

Indexers can be used when you want to treat the instances of our class, struct, or interface as arrays. An indexer is defined in a way that is similar to defining a property because you use the get and set accessors to define an indexer. However, the difference lies in the fact that while defining an indexer, we do not give it a name; instead, we refer to an indexer by using this keyword. The **this** keyword is used to refer to the instances of the class. The syntax for defining an indexer is as follows:

```
<access modifier> <return type> this [argument_list]
{
      get{}
      set{}
}
```

In the declaration of an indexer, we can use **private**, **protected**, **public**, or **internal** modifier. The return type maybe any of the valid C# data types. The **this** keyword refers to the objects of the current class while **argument_list** refers to the parameters that are passed to the indexer. Note that while defining an indexer, you must specify at least one parameter.

Now let's perform the following steps to create an application named **CreatingIndexer** that shows how you can create an indexer in a C# program:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **CreatingIndexer** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **CreatingIndexer** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.17:

Listing 4.17: Implementing an Indexer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CreatingIndexer
{
    class EmployeeList
    {
        private string[] names = new string[5];
```

```
        public string this[int index]
        {
            get
            {
                return names[index];
            }
            set
            {
                names[index] = value;
            }
        }
    }
    public class MainClass
    {
        public static void Main(string[] args)
        {
            EmployeeList el = new EmployeeList();
            el[0] = "James";
            el[1] = "Mathew";
            el[2] = "Carol";
            el[3] = "Andrew";
            el[4] = "Mary";
            Console.WriteLine("The Employee List is:");
            for (int i = 0; i <5 ; i++)
            {
                Console.WriteLine("Name of employee {0} is {1}", i+1, el[i]);
            }
            Console.ReadLine();
        }
    }
}
```

Listing 4.17 shows how an indexer can be implemented to treat the instances of the class **EmployeeList** as an array. The **EmployeeList** class has a private string array that contains five elements. We have declared an indexer using the **this** keyword. The indexer accepts an integer variable called **index** as a parameter. The **get** and **set** accessors are used to retrieve and assign names to the elements of the **EmployeeList** class. Finally, we create an instance of the class **EmployeeList** inside the **Main()** method and perform indexing on that object.

5. *Press* the **F5** key to run the **CreatingIndexer** application.

The output of the application appears, as shown in Fig.C#-4.15:

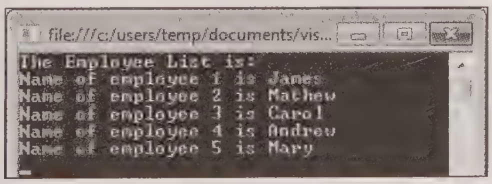


Fig.C#-4.15

Fig.C#-4.15 shows that using an indexer, the object, **el**, of the class **EmployeeList** is indexed. The **el** object is assigned some values and the same are retrieved from it. Now, let's move on to learn about encapsulation in the next section.

Implementing Encapsulation

Encapsulation is a process of hiding the internal details of an object and showing only the relevant information to a user. It is a way to organize data and methods into a structure by hiding the way an object is implemented. Encapsulation, therefore, helps to protect data from being misused and also prevents access to data by any means other than those specified.

Encapsulation is implemented through access modifiers. Access modifiers help to implement this feature by defining a scope to access data and methods in a restricted manner. Consequently, you can describe encapsulation as the ability of an object to hide its internal data and methods, and make only the intended parts of the object programmatically accessible.

Let's perform the following steps to create a console application named **EncapsulationExample** to learn how to implement encapsulation:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **EncapsulationExample** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **EncapsulationExample** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.18:

Listing 4.18: Implementing Encapsulation

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace EncapsulationExample
{
    class MyClass
    {
        private int x;
        private float y;
        protected int z;
        private void add()
        {
            Console.WriteLine("This is a private method");
        }
        public void sub()
        {
            Console.WriteLine("This is a public method");
        }
        protected void div()
        {
            Console.WriteLine("This is a protected method");
        }
    }
    class Program : MyClass
    {
        static void Main(string[] args)
        {
            Program prg = new Program();
            prg.div();
            prg.z = 20;
            prg.sub();
            Console.ReadLine();
```

```
            }
        }
    }
```

In Listing 4.18, all the class members are accessible, except the private members. Private members are accessible only in the class where they are declared. To implement encapsulation in the **MyClass** class, the **private** keyword is used to declare the data members, namely, **x**, **y**, and **z**.

5. *Press* the **F5** key to run the **EncapsulationExample** application.

   The output of the application appears, as shown in Fig.C#-4.16:

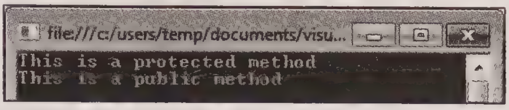


Fig.C#-4.16

Next, let's discuss about inheritance in C#.

Implementing Inheritance

The most important reason for using OOP in .NET applications is to promote the reusability of code and eliminate redundancy of code. To ensure reusability, the OOP languages promote the use of inheritance. Inheritance is defined as the property through which a child class obtains all the features defined in its parent class. A parent class is at a higher level in the class hierarchy as compared to the child class. For example, if we consider the parrot class as a child class, then it obtains its features from the parent class, that is, the bird class.

When a class inherits the common properties of another class, the class inheriting the properties is called a derived class and the class that allows inheritance of its common properties is called a base class. Inheritance in OOP is of four types:

- **Single inheritance:** Refers to the type of inheritance in which there is one base class and one derived class
- **Hierarchical inheritance:** Refers to the type of inheritance in which a base class is inherited by multiple derived classes
- **Multilevel inheritance:** Refers to the type of inheritance in which a class derives from a class, which is derived from some other class
- **Multiple inheritance:** Refers to the type of inheritance in which a derived class inherits more than one base class

C# supports single, hierarchical, and multilevel inheritances. In C#, you cannot implement multiple inheritance by directly inheriting from more than one base class in a single derived class, instead you use interfaces.

Note

Interface is a collection of data members and member functions. You learn about interface in detail later in the chapter.

Inheritance represents a kind of relationship between two classes in C#. Let's understand it through an example. Suppose there are two classes named as **Class A** and **Class B**, and the **Class B** is derived from the **Class A**, as shown in Fig.C#-4.17:

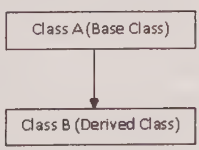


Fig.C#-4.17

Fig.C#-4.17 shows that **Class A** is referred as the base class or the parent class, and **Class B** is referred as the derived class or child class. The **Class B** derived class is a completely new class and contains all the data and methods of its base class, including its own data and methods. Now, let's discuss about the base class first.

Defining a Base Class

We know that inheritance promotes reusability of code, and it allows us to use the properties and methods of a class in some other class. The classes in which we define methods and properties that can be used or inherited by some other class are known as the base classes. You can extend the functionalities defined in the base class by inheriting them. Base classes are particularly useful for application developers, since using them can easily extend, modify, and enhance the functionality of pre-existing classes and also add some custom logic of their own.

In C#, the **base** keyword is used to access the members of the base class by a derived class. However, it is to be noted that C# supports single inheritance only. This means that your derived class can inherit from only one base class. You can define a base class similar to any other class, as shown in the following code snippet:

```
using System;

public class BaseClass
{
   public BaseClass()
   {
          Console.WriteLine("I am a Base Constructor.");
   }

   public void Hello()
   {
          Console.WriteLine("Hello! I'm a Base Class.");
   }
}
```

You can then inherit the base class **BaseClass** in a derived class using the : operator. Next, let's learn about the derived class in the following section.

Defining a Derived Class

A derived class is a class that extends the functionalities of an existing class. It automatically derives all the properties of a base class, and enables you to add more methods and properties or change them as per your requirements. A derived class can also access all the non-private data of its base class. This means that a derived class has two effective types: the type of the new class and the type of the class that it inherits.

Let's understand how you can define a derived class from a base class with the help of the following code snippet:

```
public class A
{
       public A() { }
}
public class B : A
{
       public B() { }
}
```

In the preceding code snippet, the derived class **B** inherits the properties of base class **A.**

Accessing the Base Class Members

When a class is derived from a base class, the members of the base class become the members of the derived class. The access modifier used while defining the members of the base class specifies the access status of the base class members of the derived class. The public access modifier can be used to access the members of a base class from a derived class.

Let's perform the following steps to create an application named **AccessingMembers** to learn how to access members of a base class:

1. *Repeat* the steps 1 to 3 discussed while creating **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **AccessingMembers** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **AccessingMembers** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.19:

Listing 4.19: Accessing Base Class Members

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace AccessingMembers
{
    public class BaseClass
    {
        public void BaseClassMethod()
        {
            Console.WriteLine("I am a Base Class Method()");
        }
    }
    public class DerivedClass : BaseClass
    {
        public void DerivedClassMethod()
        {
            Console.WriteLine("I am a Derived Class Method()");
            base.BaseClassMethod();
        }
    }
    public class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("I am accessing Base Class Object:");
            BaseClass bc = new BaseClass();
            bc.BaseClassMethod();
            Console.WriteLine("");
            Console.WriteLine("I am accessing Derived Class Object:");
            DerivedClass dc = new DerivedClass();
            dc.DerivedClassMethod();
            Console.Write("\nPress ENTER to quit..");
            Console.ReadLine();
        }
    }
}
```

In Listing 4.19, **BaseClass** is the parent class and **DerivedClass** is the derived class. The **DerivedClass** class inherits the features of the **BaseClass** class and is, therefore, the child class of the **BaseClass** class.

5. *Press* the **F5 key** to run the **AccessingMembers** application.

The output of the application appears, as shown in Fig.C#-4.18:

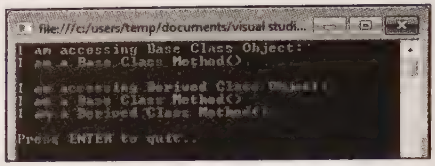


Fig.C#-4.18

In this section, you have learned how to access the members of a base class through a derived class. In the following section, you learn how to work with abstract classes.

Working with Abstract Classes

A class whose objects cannot be instantiated is called an abstract class. If you have created a base class and want to ensure that no object of the base class is created later, you can make the base class as abstract. The **abstract** keyword in a class indicates that the class cannot be instantiated and is an abstract class.

Some characteristics of an abstract class are as follows:

- Cannot be instantiate directly. This implies that you cannot create an object of the abstract class; it must be inherited.
- Contain abstract as well as nonabstract members.
- You must declare at least one abstract method in an abstract class.
- Is always declared with the public access modifier.

The basic purpose of an abstract class is to provide a common definition of the base class that can be shared by multiple derived classes.

Let's perform the following steps to create an application named **AbstractClass** to learn how to use an abstract class:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **AbstractClass** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **AbstractClass** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.20:

Listing 4.20: Accessing the Abstract Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace AbstractClass
{
	abstract class absClass
	{
		public int Add(int Num1, int Num2)
		{
			return Num1 + Num2;
		}
		public int Multiply(int Num1, int Num2)
		{
			return Num1 * Num2;
		}
	}
```

```
    class absDerived : absClass
    {
            static void Main(string[] args)
            {
                    absDerived calculate = new absDerived();
                    int added = calculate.Add(10, 20);
                    int multiplied = calculate.Multiply(10, 20);
                    Console.WriteLine("Added  :  {0},  Multiplied  :  {1}",  added,
                     multiplied);
                    Console.ReadLine();
            }
    }
}
```

In Listing 4.20, an abstract class **absClass** is defined. The **absClass** class contains two methods, namely, **Add()** and **Multiply()**, which are non-abstract. The derived class, **absDerived** class, is derived from the **absClass** class. The object **calculate** of the derived class **absDerived** is used to access the methods **Add()** and **Multiply()** of the abstract class **absClass**.

5. *Press* the **F5 key** to run the **AbstractClass** application.

The output of the application appears, as shown in Fig.C#-4.19:

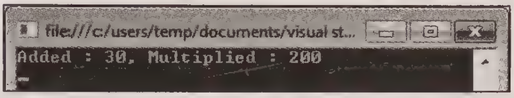


Fig.C#-4.19

Now let's learn about another type of class in C#, that is, sealed class.

Working with Sealed Classes

A sealed class implies that the class cannot be used as a base class. The main purpose of using a sealed class is to take away the inheritance feature from the users so that they cannot derive a class from a sealed class. Once you have declared a class as sealed, no other class can inherit that class. The **sealed** keyword is used to indicate that a class cannot be inherited.

Let's understand the concept of a sealed class with the help of an example given in Listing 4.21:

Listing 4.21: Example of a Sealed Class

```
sealed class MyClass
{
        //Data Members
        public void GetDetail()
        {
        }
        public void ShowDetail()
        {
        }
}
class MyMainClass
{
        //Instantiation of MyClass class
        //Method Calling
}
```

In Listing 4.21, **MyClass** is the sealed class and is instantiated from the **MyMainClass** class. Let's use the following statement to attempt to derive a class from the **MyClass** class:

```
class DerivedClass : MyClass
```

When you execute the preceding statement, you will get an error message: **DerivedClass** cannot inherit from the sealed class **MyClass**. This is because the **MyClass** base class is declared as a sealed class. Now, let's learn about polymorphism in the next section.

Implementing Polymorphism

Polymorphism, in general, can be explained as one entity containing multiple forms. In C#, you can use one procedure in multiple ways with the help of polymorphism. For example, suppose you have to write a program for calculating the area of some geometrical figure. You can use the same method name for calculating the area of a circle, a triangle, or a rectangle using different parameters.

The important features of polymorphism are as follows:

- Allows you to invoke methods of a derived class through the base class reference during run time
- Helps implement different implementations of multiple methods that are called through the same name.
- Helps call a method of a class irrespective of the specific implementation it provides.

In C#, there are two ways to implement polymorphism:

- Compile time polymorphism
- Run-time polymorphism

Let's learn about them in detail in the next section.

Implementing Compile Time Polymorphism

When the compiler compiles a program, the compiler has the information about the method arguments. Accordingly, the compiler binds the appropriate method to the respective object at the compile time itself. This process is called compile time polymorphism or early binding. You can implement compile time polymorphism through overloaded methods and operators. The arguments passed are matched in terms of number, type, and order; then the overloaded methods are invoked.

Compile time polymorphism is categorized as follows:

- Method overloading
- Operator overloading

Method Overloading

Method overloading is a concept in which a method behaves according to the number and types of parameters passed to it. Method overloading allows you to define multiple methods with the same name, but with different signatures. When you call overloaded methods, the compiler automatically determines which method should be used according to the signature specified in the method call.

Note

As described earlier, a method signature is the combination of the method's name along with the number and types of the parameters.

Let's perform the following steps to create an application named **MethodOverloading** to learn how to overload a method:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **MethodOverloading** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **MethodOverloading** application is created.

4 In the **Program.cs** file, *add* the highlighted code given in Listing 4.22:

Listing 4.22: Overloading Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MethodOverloading
{
    public class Shape
    {
        public void Area(int Side)
        {
            int SquareArea = Side * Side;
            Console.WriteLine("The Area of Square is: " + SquareArea);
        }
        public void Area(int Length, int Breadth)
        {
            int RectangleArea = Length * Breadth;
            Console.WriteLine("The Area of Rectangle is: " + RectangleArea);
        }
    }
    class MainClass
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            shape.Area(15);
            shape.Area(10, 20);
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
}
```

In Listing 4.22, the **Area()** method of the **Shape** class is overloaded to calculate the area of a square and a rectangle. When the **Area()** method is called, the compiler finds an exact match of the method, in which the data type of original parameters are the same. The retrieved method is then executed. If the compiler finds multiple matches, it generates an error message.

5 *Press* the **F5 key** to run the **MethodOverloading** application.

The output of the application appears, as shown in Fig.C#-4.20:

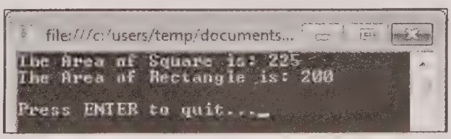


Fig.C#-4.20

Now, let's learn about another way of implementing compile time polymorphism in the following section.

Operator Overloading

All the operators have their specified meaning and functionality; for example, the + operator adds two numerals and the – operator subtracts two numerals. However, at times, you might need to change the default functionality of an operator. You can do so by operator overloading; for example, the + operator can be overloaded to concatenate two strings, instead of numerals.

The mechanism of assigning a special meaning to an operator, according to user defined data types such as classes, is known as operator overloading. It is not possible to overload all the operators. Table 4.3 shows the overloading status of different operators:

Table 4.3: Operators and Their Overloading Status

Operators	Type	Overloading Status
+, -, !, ~, ++, --, true, false	Unary	Can be overloaded
+, –, *, /, %, &, |, ^, <<, >>	Binary	Can be overloaded
==, !=, <, >, <=, >=	Comparison	Can be overloaded
&&, ||	Conditional logic	Cannot be overloaded, but can be evaluated using the & and | operators
[]	Array Indexing	Cannot be overloaded, but can help in defining indexers
()	Cast	Cannot be overloaded, but can help in defining new conversion operators
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Assignment	Cannot be overloaded, but += operator, for example, is evaluated using + operator, which can be overloaded
=, ., ?:, ->, new, is, sizeof, typeof	Other	Cannot be overloaded

Let's perform the following steps to create an application named **OperatorOverloading** to learn how to overload a method:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **OperatorOverloading** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **OperatorOverloading** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.23:

Listing 4.23: Overloading an Operator

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace OperatorOverloading
{
    class UnaryOperator
    {
        private int Number1;
        private int Number2;
        public UnaryOperator()
        {
        }
        public UnaryOperator(int number1, int number2)
        {
            Number1 = number1;
            Number2 = number2;
        }
        public void ShowData()
```

```
        {
                Console.WriteLine("The Numbers are: " + Number1 + " and " +
Number2);
                ;
        }
        public static UnaryOperator operator -(UnaryOperator opr)
        {
                UnaryOperator obj = new UnaryOperator();
                obj.Number1 = -opr.Number1;
                obj.Number2 = -opr.Number2;
                return obj;
        }
    }
    class MainClass
    {
        public static void Main()
        {
                UnaryOperator opr1 = new UnaryOperator(20, 30);
                Console.WriteLine("Before Operator Overloading");
                opr1.ShowData();
                UnaryOperator opr2 = new UnaryOperator();
                opr2 = -opr1;
                Console.WriteLine("---------------------------");
                Console.WriteLine("After Operator Overloading");
                opr2.ShowData();
                Console.Write("\nPress ENTER to quit...");
                Console.ReadLine();
        }
    }
}
```

In Listing 4.23, the **–()** operator method takes one argument of type **UnaryOperator** and changes the sign of data member **opr**.

5 *Press* the **F5 key** to run the **OperatorOverloading** application.

The output of the application appears, as shown in Fig.C#-4.21:

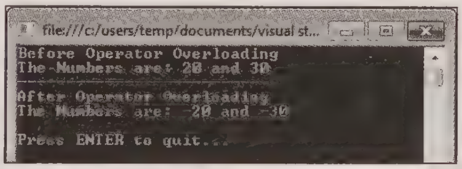


Fig.C#-4.21

This completes our discussion on how to implement compile-time polymorphism. Now, let's move on to the next section and examine how run-time polymorphism is implemented.

Implementing Run-Time Polymorphism

Run-time polymorphism in C# is better known as overriding. Overriding allows a derived class to define a specific implementation for a method, other than the implementation defined by its base class. The implementation of the method in a derived class overrides or replaces the implementation of the method in its base class. This feature is known as runtime polymorphism because the compiler binds the method to an object while the program is being executed (runtime), and not when the program is being compiled.

When you call a method, the method defined in the derived class is invoked and executed instead of the one in the base class, but with the following conditions:

- You must declare the base class method as **virtual**
- You must implement the derived class method using the **override** keyword

Let's now learn how to override a method in C#.

Overriding a Method

A basic concept behind OOP is that you can create virtual methods, which can be overridden in a derived class. OOP allows the derived class to provide implementation of a method that is defined in the parent class in C#. You can do this in C# with the help of the **override** and **virtual** keywords. To do this, you must explicitly define the methods in the base class as virtual. You use the **virtual** keyword in a method to indicate that you want to have a base method overridden in a derived class. Using the **override** keyword, you must specifically tell the compiler that you are intending to override an existing virtual method.

Let's perform the following steps to create an application named **MethodOverriding** to learn how you can override a method in a derived class by defining a virtual method in the base class:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **MethodOverriding** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **MethodOverriding** application is created
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.24:

Listing 4.24: Overriding a Method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MethodOverriding
{
      class Program
      {
              static void Main(string[] args)
              {
                      Person p = new Person();
                      p.setAge(18);
                      AdultPerson ap = new AdultPerson();
                      ap.setAge(18);
                      Console.WriteLine("Person Age: {0}", p.getAge());
                      Console.WriteLine("AdultPerson Age: {0}", ap.getAge());
             Console.Write("\nPress ENTER to quit...");
             Console.ReadLine();
         }
    }
    public class Person
    {
        private int fAge;
        public Person()
        {
            fAge = 21;
        }
        public virtual void setAge(int age)
        {
            fAge = age;
        }
```

```
            public virtual int getAge()
            {
                return fAge;
            }
        }
        public class AdultPerson : Person
        {
            public AdultPerson() { }
            override public void setAge(int age)
            {
                if (age > 21)
                    base.setAge(age);
            }
        }
    }
```

In Listing 4.24, two virtual methods, namely, **setAge()** and **getAge(),** are defined inside the class **Person**. The class **AdultPerson** inherits the class **Person** and overrides the method **setAge()**. The if statement inside the **setAge()** method of the derived class **AdultPerson** checks whether the age is greater than **21**. If the age is greater than **21,** then the **setAge()** method of the base class **Person** is called to set the age as per the value passed in the argument. Inside the **Main()** method of the **Program** class, the **setAge()** method from the base class **Person** and derived class **AdultPerson** is called.

5. *Press* the **F5** key to run the **MethodOverriding** application.

The output of the application appears, as shown in Fig.C#-4.22:

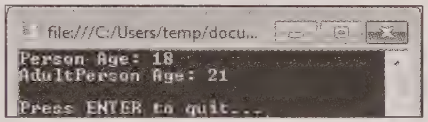


Fig.C#-4.22

In Fig.C#-4.22, you can see that the **setAge()** method of the base class **Person** sets the age to **18** and displays it. However, the **setAge()** method of the derived class **AdultPerson** does not display the age **18** as it is smaller than **21**.

Next, let's discuss how to work with interfaces to implement multiple inheritance.

Working with Interfaces

Interface is a collection of abstract data members such as methods, events, and properties. Interfaces in C# are introduced to provide the feature of multiple inheritance to classes in C#. The methods defined in an interface do not have their implementation and only specify the parameters they will take and the type of values they will return. An interface is always implemented in a class.

Interface in C# is equivalent to an abstract base class. You cannot instantiate an object through an interface, but you can offer a set of functionalities that is common to several different classes.

Let's learn how you can define, implement, and inherit an interface in C#.

Defining an Interface

An interface is declared in a manner similar to that of declaring a class. The difference is that a class is declared with the **class** keyword and an interface is declared with an **interface** keyword. The syntax of defining an interface in C# is as follows:

```
interface MyInterface
{
    //Abstract Method Declarations in Interface Body
}
```

In the preceding syntax, the **interface** keyword defines an interface named **MyInterface**.

You can declare an interface, as shown in the following code snippet:

```
interface MyFirstInterface
{
        void MyFirstMethod();
}
interface MySecondInterface
{
        void MySecondMethod();
}
```

In the preceding code snippet, two interfaces, **MyFirstInterface** and **MySecondInterface,** are declared. The **MyFirstInterface interface** contains the declaration of a method **MyFirstMethod,** and the **MySecondInterface** interface contains the declaration of a method **MySecondMethod**. You can implement the **MyFirstInterface** and **MySecondInterface** interfaces in a class or a struct and define the implementation of the methods **MyFirstMethod** and **MySecondMethod**.

Implementing an Interface

You can implement interface in C# using a class that implements the features of the interface. The implementation of interface is similar that of inheriting a class. The code snippet for implementing an interface is given in Listing 4.25:

Listing 4.25: Showing the Code to Implement an Interface

```
class Program : MyFirstInterface, MySecondInterface
    {
        public void MyFirstMethod()
        {
            Console.WriteLine("Inside Method1");
        }
        public void MySecondMethod()
        {
            Console.WriteLine("Inside Method2");
        }
    }
    class InterfaceTest
    {
       public static void Main(string[] args)
        {
            Program p = new Program();
            p.MyFirstMethod();
            p.MySecondMethod();
            Console.ReadLine();
        }
    }
```

In Listing 4.25, the **Program** class implements two interfaces named, **MyFirstInterface** and **MySecondInterface**. The methods **MyFirstMethod** and **MySecondMethod** of the **MyFirstInterface** and **MySecondInterface** interfaces, respectively, are also defined within the **Program** class. The object **p** of **Program** class is used to call the methods **MyFirstMethod** and **MySecondMethod** inside the **Main()** method of the **InterfaceTest** class.

Implementing Interface Inheritance

You know that an interface in itself provides no functionality. You also know that an interface is implemented by a class or a struct to provide a complete implementation for all the members, such as methods, properties, events, or indexers, declared in the interface. If you implement an interface in a base class, which is inherited by a derived class, then the derived class automatically inherits the implementation of the interface in the base

class. In C#, a class can implement more than one interface, and an interface can also be implemented by another interface.

Let's perform the following steps to create an application named **InterfaceInheritance** to learn to implement an interface inheritance:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **InterfaceInheritance** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **InterfaceInheritance** application is created.
4. In the **Program.cs** file, *add* the highlighted code given in Listing 4.26:

Listing 4.26: Implementing an Interface Inside a Class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceInheritance
{
      interface BaseInterface
      {
              void GetDetail();
      }
      interface DerivedInterface : BaseInterface
      {
              void ShowDetail();
      }
      class InterfaceImplementer : DerivedInterface
      {
              string Name;
              int Age;
              string Address;
              long PhoneNumber;
              public void GetDetail()
              {
                      Console.Write("Enter Your Name:");
                      Name = Console.ReadLine();
                      Console.Write("Enter Your Age:");
                      Age = int.Parse(Console.ReadLine());
                      Console.Write("Enter Your Address:");
                      Address = Console.ReadLine();
                      Console.Write("Enter Your Phone Number:");
                      PhoneNumber = long.Parse(Console.ReadLine());
              }
              public void ShowDetail()
              {
                      Console.WriteLine("");
                      Console.WriteLine("Your Details are:");
                      Console.WriteLine("Name: " + Name);
                      Console.WriteLine("Age: " + Age);
                      Console.WriteLine("Address : " + Address);
                      Console.WriteLine("Phone Number: " + PhoneNumber);
              }
      }
      class MainClass
```

```
    {
        static void Main(string[] args)
        {
            InterfaceImplementer MyObj = new InterfaceImplementer();
            MyObj.GetDetail();
            MyObj.ShowDetail();
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
}
```

In Listing 4.26, an interface, that is, **BaseInterface,** is declared. It contains the declaration of a method, that is, **GetDetail()**. Another interface called **DerivedInterface** is declared, which inherits the interface **BaseInterface**. The interface **DerivedInterface** contains the declaration of the method **ShowDetail()**. A class called **InterfaceImplementor** that implements the **DerivedInterface** interface is defined. The methods **GetDetail()** and **ShowDetail()** are defined inside the class **InterfaceImplementor**. The **GetDetail()** and **ShowDetail()** methods are called inside the **Main()** method of the **MainClass** by the object **MyObj** of the class **InterfaceImplementor**.

5. *Press* the **F5 key** to run the **InterfaceInheritance** application.

   The output of the application appears, as shown in Fig.C#-4.23:

Fig.C#-4.23

As seen in Fig.C#-4.23, you are prompted to enter your name, as the **GetDetail()** method is called to retrieve your details.

6. *Enter* your name and *press* the **Enter** key. In our case, we have entered the name as **Student**, as shown in Fig.C#-4.24:

Fig.C#-4.24

7. Similarly, *enter* your age, address, and phone number, as shown in Fig.C#-4.25:

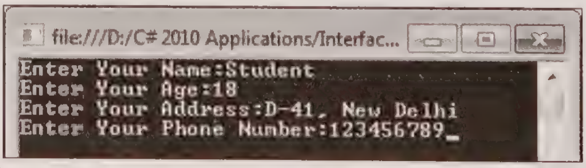


Fig.C#-4.25

After you enter all the details, the output appears on the screen, as shown in Fig.C#-4.26:

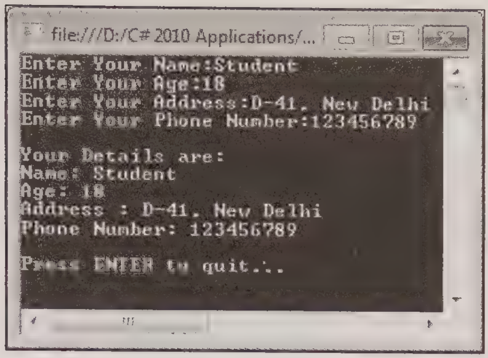


Fig.C#-4.26

Now, let's learn about the namespaces in C#.

Working with Namespaces

The concept of namespace is not new in C#. The concept of namespaces has been used in C++ and Java since a long time. A namespace is a kind of wrapper around one or more structural elements that make the elements unique.

Whether or not you explicitly declare a namespace in the C# source file, the compiler adds a default namespace in your C# application. Namespaces have public access and is not modifiable.

A namespace in C# has the following properties:

- It organizes large code projects
- The · operator delimits it
- The using directive means that you do not need to specify the name of the namespace for every class

Namespaces in C# are of two categories: user-defined and system-defined namespaces. The user-defined namespaces are the namespaces you create in the code, and the system-defined namespaces are the one that are already added in your code when you create a new application.

All the codes you write exist within an implied namespace that exists for the current context of the code. The **using** directive is used to tell the compiler which namespaces you want to use in the program. The most common use of the **using** directive is to include the **System** namespace in your program, so that you can use the maximum functionalities of the **System** namespace.

Let's learn to create a user-defined namespace and also how you can pass the reference of the namespace in your program in the following sections.

Note

Namespaces are always public; therefore, the declaration of namespace cannot include any access modifiers.

Next, let's learn how to create namespaces in a C# application.

Creating Namespaces

When you create a large number of classes, it can be helpful to categorize them into their own namespaces for the better organization of code. You can use namespaces to group the type so that you can use it multiple times and also avoid the conflict with the names that are already declared. When you create a namespace, you must use the **namespace** keyword, which is followed by its name.

Let's create an application named **MyNamespace** to learn to create namespace by performing the following steps:

1. *Repeat* the steps 1 and 2 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. In the **New Project** dialog box, *select* the **Visual C#** option from the **Installed Templates** pane and the **Class Library** option from the middle pane.
3. *Enter* **MyNamespace** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
4. *Click* the **OK** button. The **MyNamespace** application is created.
5. In the **Class1.cs** file, *add* the highlighted code given in Listing 4.27:

Listing 4.27: Creating Namespace

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyNamespace
{
	public class MyClass
	{
		public static void MyMethod()
		{
			Console.WriteLine("I am using my Namespace");
		}
	}
}
```

Listing 4.27 shows that you are creating a namespace named **MyNamespace**, which consists of a class named as **MyClass**.

6. *Select* the **Build →Build Solution** option from the menu bar to build the **MyNamespace** application.

In the next section, you learn how to reference a user-defined namespace.

Referencing Namespaces

You can also use a user-defined namespace in your application. To use a user-defined namespace in your application, you must add reference of that namespace to your application.

Let's create an application named **MyApplication** to learn to add reference of **MyNamespace** namespace by performing the following steps:

1. *Repeat* the steps 1 to 3 discussed while creating the **CreatingMethod** application in the **Defining a Method** section of this chapter.
2. *Enter* **MyApplication** in the **Name** text box to specify the name of the application, and *specify* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **MyApplication** application is created.
4. *Right-click* the application name, **MyApplication**, in the **Solution Explorer** pane (Fig.C#-4.27).
5. *Select* the **Add Reference option** from the context menu, as shown in Fig.C#-4.27:

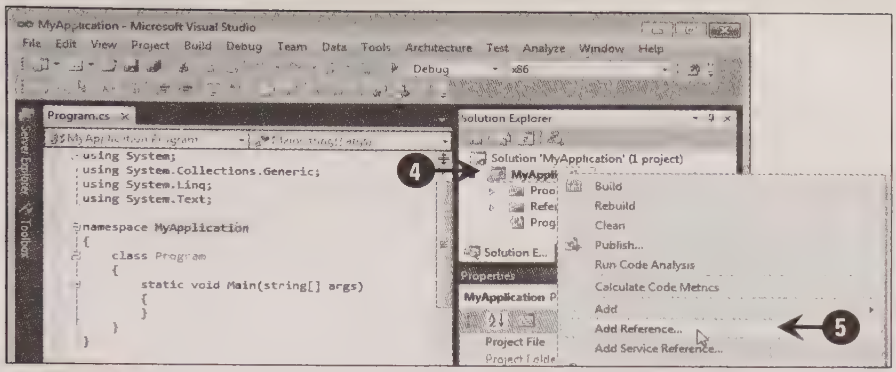


Fig.C#-4.27

The **Add Reference** dialog box opens (Fig.C#-4.28).

6 In the **Add Reference** dialog box, click the **Browse** tab to locate the **MyNamespace.dll** file (Fig.C#-4.28).

7 *Browse* to the **MyNamespace.dll** file in the **MyNamespace** project (Fig.C#-4.28).

8 *Click* the **OK button,** as shown in Fig.C#-4.28:

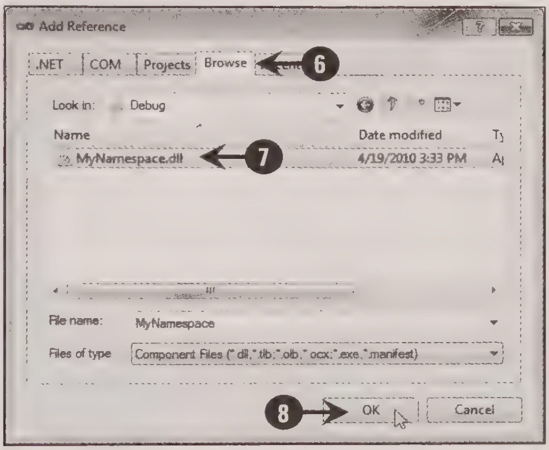


Fig.C#-4.28

The **Class1.cs** file is added to the **MyApplication** project.

9 In the **Program.cs** file of the **MyApplication** project, *add* the highlighted code given in Listing 4.28:

Listing 4.28: Adding Reference

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using MyNamespace;
namespace MyApplication
{
    class Program
    {
        static void Main(string[] args)
```

```
                {
                    MyClass.MyMethod();
                    Console.Write("\n\nPress ENTER to quit...");
                    Console.ReadLine();
                }
            }
        }
```

Listing 4.28 shows that a user-defined namespace, that is, **MyNamespace**, is added to the **MyApplication** project.

10 *Press* the **F5** key to run the **MyApplication** application.

The output of the application appears, as shown in Fig.C#-4.29:

Fig.C#-4.29

With this, you have come to the end of this chapter. Let's now summarize what you have learned in this chapter.

Summary

In this chapter, you have learned about:

- Defining classes and objects in a C# application
- Garbage collection mechanism in .NET Framework
- Using properties and indexers in a C# application
- Hiding irrelevant information in a class using encapsulation
- Implementing reusability of code through inheritance
- Defining polymorphism and its implementation at compile time and runtime
- Creating data types that store small amount of data with the help of structures
- Creating properties in a C# application
- Specifying the members that must be supplied by classes with the help of interfaces
- Organizing C# code with the help of namespaces

Chapter 5

Programming with Windows Forms Controls

In this Chapter:

- Performing Common Operations on Form
- Handling Common Events for Windows Forms Applications
- Working with Windows Forms Controls

Designing user interfaces (UIs) for applications is one of the most important and crucial tasks. User interacts and uses the functionality of an application through its UI. A UI contains visually appealing interface with controls that can be customized as per user requirements. Apart from Console applications that run directly from Command Prompt, other applications, such as Windows Forms applications and Windows Presentation Foundation (WPF) applications designed on .NET Framework are built using forms. A form is a building block of an application and represents visual surface on which different Windows controls, such as Buttons and Labels are displayed to the user. You can build a Windows Forms application by adding controls to forms and customizing their behavior by using any .NET compliant language, such as C# and Visual Basic .NET (VB .NET). In other words, a form acts as a container for .NET controls. In addition, it offers various features, such as tabbed navigation (the feature by which a user can navigate to all the controls in a cycle), ordering of tabs, and listening to mouse events.

You can customize a form by specifying its characteristics, such as size, location, appearance, and fore color by using the **Form** class properties, such as **Size**, **Location**, **BackColor**, and **ForeColor.** Different customizations can be applied to forms so that they represent different visual effects for different users. Methods such as **SetDesktopLocation()** can be used to define the position of a form on the desktop. You can also define events, such as **Activated** to perform some modifications to a control, while the control is in active state.

In this chapter, you learn about the common operations that can be performed on a form, such as changing the title of a form and adding controls to a form. In addition, you learn to handle mouse and keyboard events for Windows Forms application. Finally, you learn to add different controls to a form and use them.

Now, let's first learn about the common operations that can be performed on a form in the following section.

Performing Common Operations on Form

As stated earlier, a form constitutes a part of Microsoft .NET Framework and has a Graphical User Interface (GUI) for building Windows Forms applications. You can add different types of controls, such as **Button, Label,** and **RichTextBox** to a form to create Windows Forms applications. A control can be added to a form from Toolbox by either dragging and dropping it or double-clicking it.

Note

A Toolbox consists of icons of all the controls that can be added on a form. You can drag and then drop the controls on a form or simply double-click a control to add it to a form. To open Toolbox, select View→Toolbox.

Let's discuss the following basic operations that can be performed on a form:

- Changing the title of a form
- Showing and hiding the Maximize, Minimize, and Close buttons
- Specifying the initial state and position of a form
- Creating a multiform Windows Forms application
- Setting the startup form in a multiform Windows Forms application
- Using the MessageBox.Show() Method
- Adding a control to a form
- Anchoring and docking a control
- Enabling and disabling controls
- Specifying the tab order of controls

Let's first learn to change the title of a form.

Changing the Title of a Form

The default title of the form in a Windows Forms application is set to Form1. However, you can change the title of a form at design time as well as at run time by changing the Text property of the form.

Let's create a Windows Forms application, **SampleApplication,** in which you can change the title of the **Form1** form by performing the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from menu bar in Visual Studio 2010 IDE. The **New Project** dialog box appears.
3. *Select* **Visual C#→Windows** in the **Installed Templates** pane and **Windows Forms Application** option in the middle pane in the New Project dialog box.
4. *Enter* **SampleApplication** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
5. *Click* the **OK** button. The **SampleApplication** application is created with the specified name and location.
6. *Click* anywhere on the **Form1** form and *press* the **F4** key to open its **Properties** window.
7. *Set* the **Text** property of the **Form1** form to **FirstForm** in the **Properties** window (Fig.C#-5.1).

   After changing the Text property of the **Form1** form, its title appears, as shown in Fig.C#-5.1:

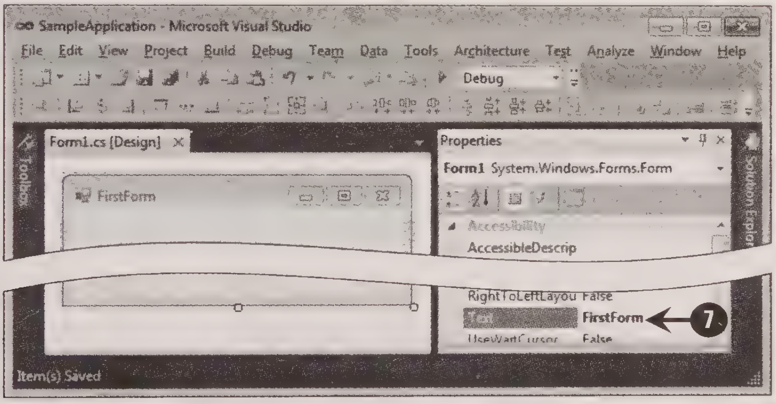


Fig.C#-5.1

In Fig.C#-5.1, the title of the **Form1** form is changed to **FirstForm**.

Note

You can also change the Text property of Form1 by entering the following code snippet at the Load event of Form1:
this.Text = "FirstForm";
The preceding code snippet changes the title of Form1 to FirstForm.

Next, let's learn how to show and hide the Maximize, Minimize, and Close buttons on a Windows form.

Showing and Hiding the Maximize, Minimize, and Close Buttons

A form contains certain buttons, such as Minimize, Maximize, and Close. These buttons are present at the upper-right corner of the form and are visible by default. You can, if required, hide these buttons by setting the **ControlBox** property of the Form1 form to **False**.

Let's perform the following steps in the **SampleApplication** application created in the **Changing the Title of a Form** section of this chapter, to hide Maximize, Minimize, and Close buttons:

1. *Set* the **ControlBox** property of the **FirstForm** form to **False** in the **Properties** window to hide the **ControlBox**, as shown in Fig.C#-5.2:

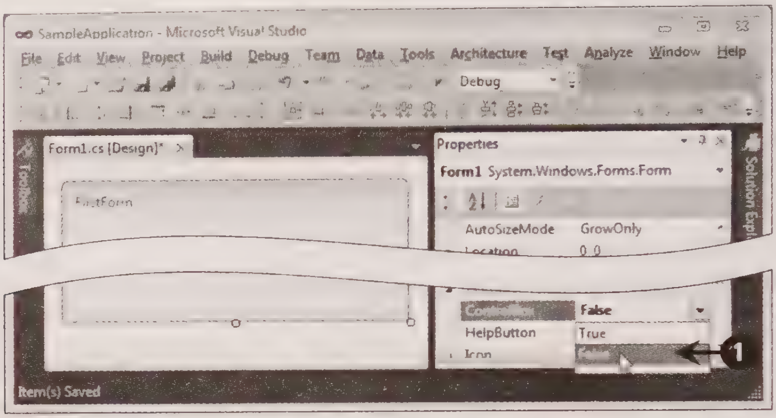


Fig.C#-5.2

2. *Press* the **F5** key to run the **SampleApplication** application. The output appears, as shown in Fig.C#-5.3:

Fig.C#-5.3

In Fig.C#-5.3, the **Minimze**, **Maximize**, and **Close** buttons are not visible in the **FirstForm** form. Now, let's learn to specify the initial state and position of a form.

Specifying the Initial State and Position of a Form

A form can be displayed on the screen of a monitor or a projector in the following modes:

- **Maximized state:** Occupies the full display area of the screen.
- **Minimized state:** Appears minimized to the taskbar.
- **Normal state:** Appears normally with the same size and location as specified in the **Properties** window. This is the default state of a form.

You can set the initial state of a form by setting its **WindowState** property. Initial state of a form refers to the state (maximized, minimized, or normal) in which the form is displayed for the first time. Perform the following steps in the **SampleApplication** application to maximize a Windows form:

1. *Set* the **WindowState** property of the **FirstForm** form to **Maximized** in the **Properties** window, as shown in Fig.C#-5.4:

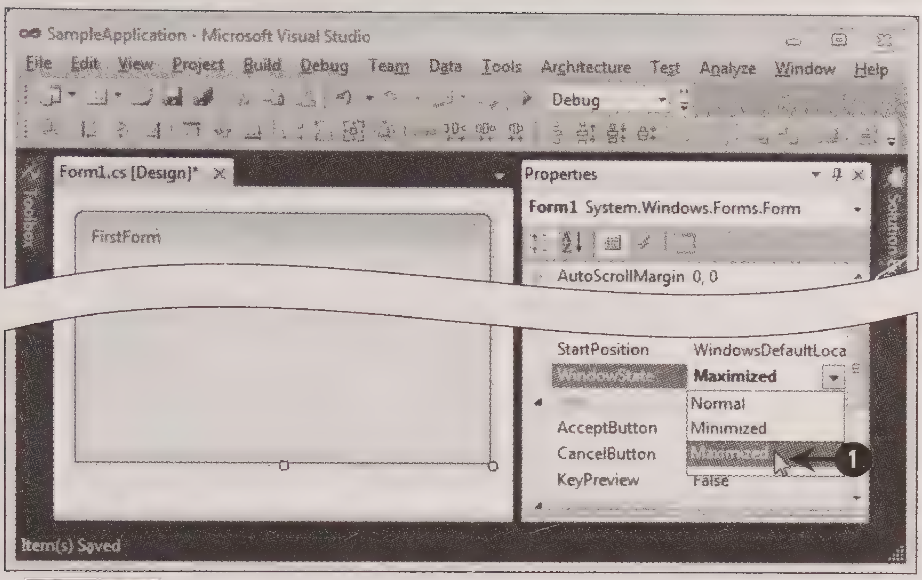


Fig.C#-5.4

In Fig.C#-5.4, the **WindowState** property of the **FirstForm** form is **Maximized**.

Note

The default value of the WindowState property is Normal.

Sometimes, you may need to display a form at a certain position. You can make a form appear at a specific position on the screen with the help of its **StartPosition** property. The **StartPosition** property of the **Form1** form takes different values that are listed in Table 5.1:

Table 5.1: Representing Possible Values for StartPosition Property

Value	Description
CenterParent	Centers the form within the bounds of its parent form
CenterScreen	Centers the form with specified dimensions of the form using its **Size** property
Manual	Determines the starting position of the form using its **Location** and **Size** properties
WindowsDefaultBounds	Positions the form at the default location of the Windows interface and has the bounds determined by Windows default
WindowsDefaultLocation	Positions the form at the default location of the Windows interface and has the dimensions specified in the **Size** property of the form

In the **SampleApplication** application, set the **StartPosition** property of the **FirstForm** form to **CenterScreen** in the **Properties** window, as shown in Fig.C#-5.5:

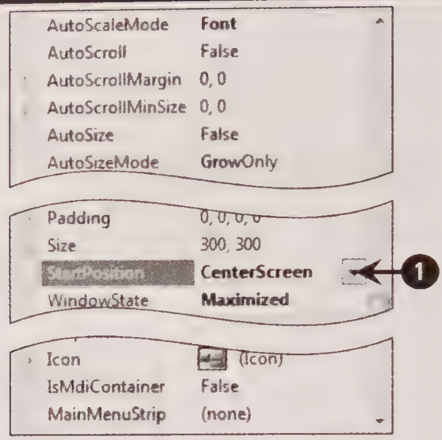


Fig.C#-5.5

As shown in Fig.C#-5.5, the **StartPosition** property of the **SampleApplication** application is set to **CenterScreen**.

Creating a Multiform Windows Forms Application

In some cases, you may want to add more than one form to your Windows Forms application. A multiform Windows application allows you to add one or more forms to your application and use the data of a form in another form. You can add multiple forms to a single Windows Forms application.

Let's perform the following steps to add another form to the **SampleApplication** application:

1. *Right-click* the **SampleApplication** application in Solution Explorer and *select* **Add→New Item** from context menu, as shown in Fig.C#-5.6:

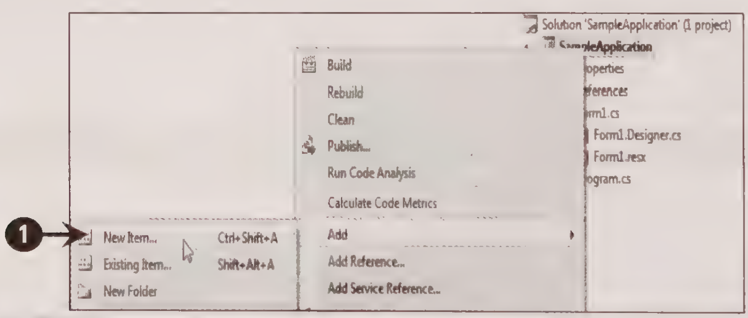


Fig.C#-5.6

The **Add New Item** dialog box opens (Fig.C#-5.7).

2. *Select* the **Windows Form** option in the Add New Item dialog box (Fig.C#-5.7).
3. *Enter* **SecondForm.cs** in the **Name** text box (Fig.C#-5.7).
4. *Click* the **Add** button, as shown in Fig.C#-5.7:

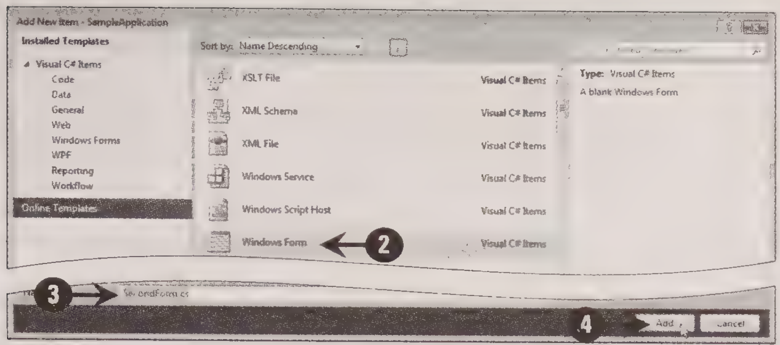


Fig.C#-5.7

A new form named **SecondForm** is added to the **SampleApplication** application, as shown in Fig.C#-5.8:

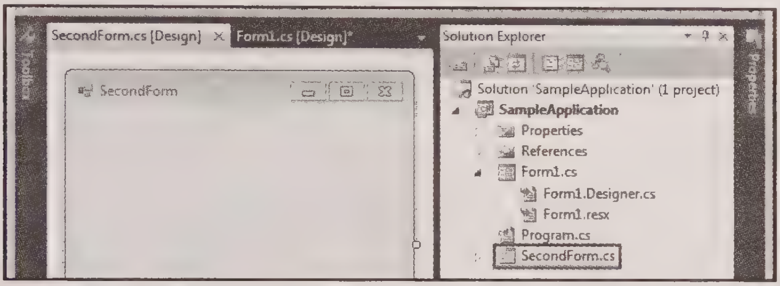


Fig.C#-5.8

In the same way, you can add more forms to your application.

Setting the Startup Form in a Multiform Windows Forms Application

Whenever you open a multiform Windows application, the **Form1** form appears as the default startup form; however, you can also set any other form as the startup form. In C#, you can set the startup form in the **Program.cs** file.

Perform the following steps in the **SampleApplication** application to set the **SecondForm** form as the startup form in a multiform Windows application:

1. *Double-click* the **Program.cs** file in Solution Explorer. The Code Editor appears with the code, as shown in Listing 5.1:

Listing 5.1: Showing the Code of Program.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace SampleApplication
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
```

```
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

In Listing 5.1, the last line in the **Main()** method is a directive for the startup form. In our case, the startup form is **Form1**.

2. *Change* the directive for startup to the **SecondForm form**, as shown in the following code snippet:

```
Application.Run(new SecondForm());
```

3. *Press* the **F5** key to run the **SampleApplication** application. The output appears, as shown in Fig.C#-5.9:

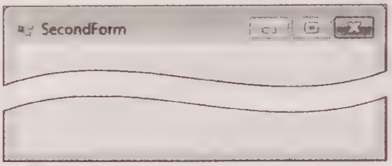


Fig.C#-5.9

In Fig.C#-5.9, the second form, **SecondForm** appears first.

Using the MessageBox.Show() Method

In C# 2010, the **MessageBox.Show()** method of .NET Framework is used to display message boxes. The **MessageBox.Show()** method is an overloaded method and its syntax is as follows:

```
public static DialogResult Show(
string text,
string caption,
MessageBoxButtons buttons,
MessageBoxIcon icon,
MessageBoxDefaultButton defaultButton,
MessageBoxOptions options,
string helpFilePath,
HelpNavigator navigator,
Object param
)
```

The different parameters of the **MessageBox.Show()** method are explained as follows:

- **text:** Displays text in message box.
- **caption:** Displays text in the title bar of the message box.
- **buttons:** Specifies which buttons to display in the message box. It is one of the **MessageBoxButton** enumeration values.
- **icon:** Specifies the icon to be displayed in the message box. It is one of the **MessageBoxIcon** enumeration values.
- **defaultButton:** Specifies the default button for the message box. It is one of the **MessageBoxDefaultButton** enumeration values.
- **options:** Specifies the display and associated options that are used for the message box. It is one of the **MessageBoxOptions** enumeration values.
- **helpFilePath:** Represents the path and name of the Help file to be displayed when a user clicks the **Help** button.

- **navigator:** Refers to one of the **HelpNavigator** values.
- **param:** Displays the numeric **ID** of the Help topic when a user clicks the **Help** button.

The following are the values of the **MessageBoxButtons** enumeration:

- **AbortRetryIgnore:** Shows the **Abort**, **Retry**, and **Ignore** buttons
- **OK:** Shows the **OK** button
- **OKCancel:** Shows the **OK** and **Cancel** buttons
- **RetryCancel:** Shows the **Retry** and **Cancel** buttons
- **YesNo:** Shows the **Yes** and **No** buttons
- **YesNoCancel:** Shows the **Yes**, **No**, and **Cancel** buttons

The following are the values of the **MessageBoxIcon** enumeration:

- **Asterisk:** Shows an icon displaying a lowercase letter i in a circle
- **Error:** Shows an icon displaying a white X in a circle with a red background
- **Exclamation:** Shows an icon displaying an exclamation in a triangle with a yellow background
- **Hand:** Shows an icon displaying a white X in a circle with a red background
- **Information:** Shows an icon displaying a lowercase letter i in a circle
- **None:** Shows no icons
- **Question:** Shows an icon displaying a question mark in a circle
- **Stop:** Shows an icon displaying a white X in a circle with a red background
- **Warning:** Shows an icon displaying an exclamation in a triangle with a yellow background

The following are the values of the **MessageBoxDefaultButton** enumeration:

- **Button1:** Makes the first button on message box the default button
- **Button2:** Makes the second button on message box the default button
- **Button3:** Makes the third button on message box the default button

The following are the options of the **MessageBoxOptions** enumeration:

- **ServiceNotification:** Displays message box on the active desktop when a service makes a call notifying the user about an event. A message box is displayed on the current active desktop, even if there is no user logged on the computer.
- **DefaultDesktopOnly:** Displays a message box on the active desktop.

  The **DefaultDesktopOnly** option is the same as the **ServiceNotification** option except that it displays a message box in the interactive window station.

Note

An interactive window station is automatically created when a user logs on to a computer system. It can display a user interface and accept some input from the user as well.

- **RightAlign:** Displays the message box text as right-aligned.
- **RtlReading:** Specifies that the message box text is displayed in the right to left reading order.

The result of the **Show()** method of the **MessageBox** class is a value from the **DialogResult** enumeration. The following are the buttons for displaying the result of the **Show()** method using the **DialogResult** enumeration:

- **Abort:** Returns **Abort** when a button named **Abort** is clicked on message box.
- **Cancel:** Returns **Cancel** when a button named **Cancel** is clicked on message box
- **Ignore:** Returns **Ignore** when a button named **Ignore** is clicked on message box
- **No:** Returns **No** when a button named **No** is clicked on message box

- **None:** Returns nothing from message box
- **OK:** Returns **OK** when a button named **OK** is clicked on message box
- **Retry:** Returns **Retry** when a button named **Retry** is clicked on message box
- **Yes:** Returns **Yes** when a button named **Yes** is clicked on message box

Let's perform the following steps in the **SampleApplication** application to learn how to use the **MessageBox.Show()** method:

1. *Drag* a **Button** control and a **TextBox** control from Toolbox and drop it on the **SecondForm** form, as shown in Fig.C#-5.10:

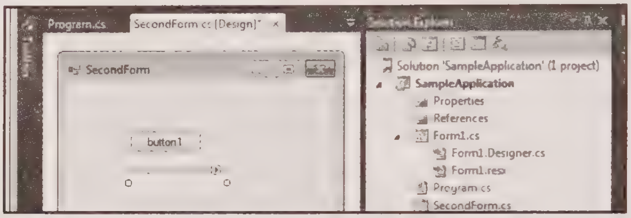


Fig.C#-5.10

In Fig.C#-5.10, the **Button** and **TextBox** controls that are added to the **SecondForm** form are displayed.

Note

You learn more about adding controls to a form later in the chapter.

2. *Double-click* the **Button** control on the **SecondForm** form to generate the **Click** event of **button1**.

Note

You learn about generating events to Windows controls later in the chapter.

3. *Add* the code shown in Listing 5.2 on the **Click** event of **button1**:

Listing 5.2: Adding Code on button1_Click Event

```
if (textBox1.Text == "")
  {
        MessageBox.Show("You must enter a name.", "Name Entry Error",
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
  }
```

4. *Press* the **F5** key to run the **SampleApplication** application. The form, **SecondForm** appears (Fig.C#-5.11).
5. *Click* the **button1** button, as shown in Fig.C#-5.11:

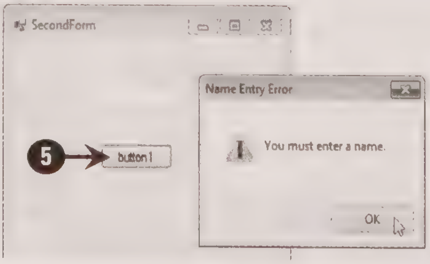


Fig.C#-5.11

In Fig.C#-5.11, a message box appears, displaying a message.

Adding a Control to a Form

The fundamental task of a form is to contain various controls, which a user can use to provide input and get the output from an application. In C# 2010, you can add various controls, such as Label, Button, and TextBox to an application from **Toolbox**.

Let's perform the following step to add a **Button** control to the **FirstForm** form:

1. *Drag* a **Button** control from **Toolbox** and *drop* it on the **FirstForm** form, as shown in Fig.C#-5.12:

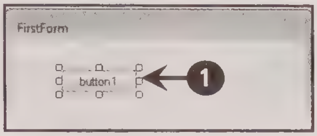


Fig.C#-5.12

In Fig.C#-5.12, the **FirstForm** form appears with the button, **button 1**.

Anchoring and Docking a Control

Docking and anchoring refer to the techniques used to align and arrange the existing controls of a form. Docking refers to attaching a control to the edge of a parent control and anchoring refers to specifying the distance that each edge of a control should maintain from the edges of the parent control.

Let's perform the following steps to dock the **Button** control in the **SampleApplication** application:

1. *Select* the **Button** control on the **FirstForm** form (Fig.C#-5.13).
2. *Set* the **Dock** property of the **Button** control in the **Properties** window, as shown in Fig.C#-5.13:

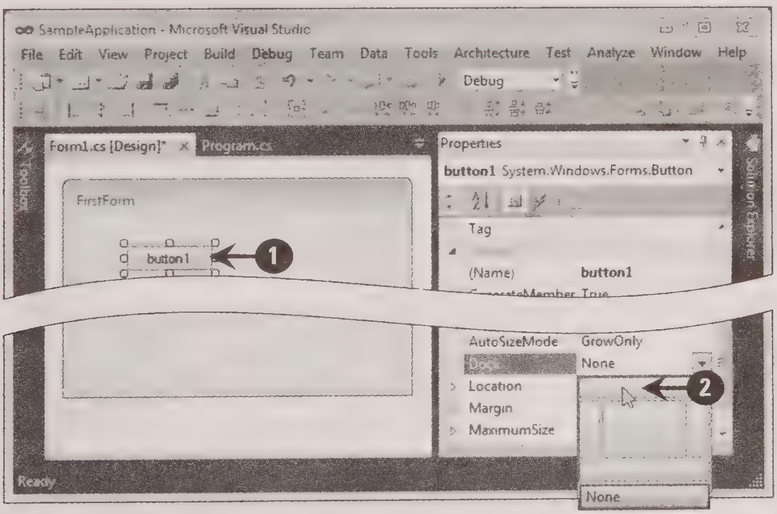


Fig.C#-5.13

The **Button** control is docked to the top edge of the **FirstForm** form, as shown in Fig.C#-5.14:

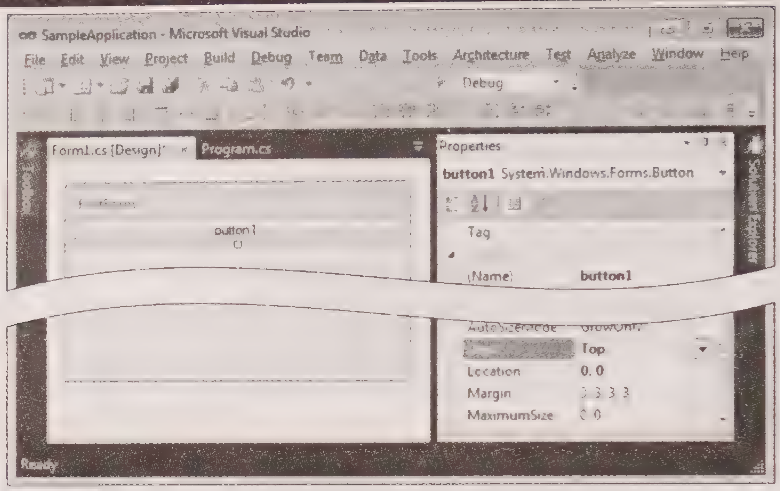


Fig.C#-5.14

You can also anchor a control to the edges of its container with the help of **Anchor** property. Let's perform the following steps to anchor the **Button** control along all the edges of the **FirstForm** form:

1. *Select* the **Button** control on the **FirstForm** form (Fig.C#-5.15).
2. *Set* the **Anchor** property of the **Button** control in the **Properties** window and click all the edges, which are seen as blank to anchor the **Button** control to all the edges of the **FirstForm** form, as shown in Fig.C#-5.15:

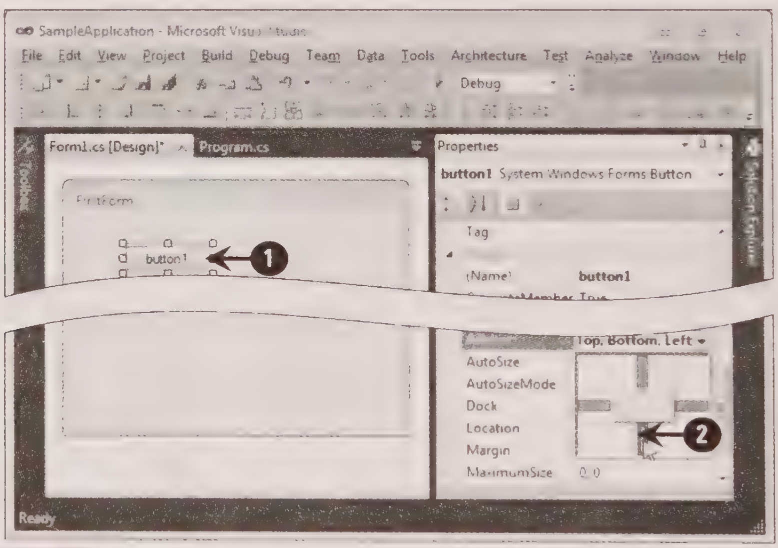


Fig.C#-5.15

3. *Press* the **F5** key to run the **SampleApplication** application. The **FirstForm** form appears, as shown in Fig.C#-5.16:

Fig.C#-5.16

In Fig.C#-5.16, the **button1** button control is anchored to all the edges of the **FirstForm** form.

Note

When you have multiple forms in a Windows Forms application, you must set one of the forms as the startup form, so that when you execute the application, the form set as the startup form is displayed. In SampleApplication, set the form, Form1 as the startup form in the Program.cs file.

4. *Resize* the **FirstForm** form from the bottom-right corner.

   The size of the **Button** control increases or decreases as you resize the **FirstForm** form, as shown in Fig.C#-5.17:

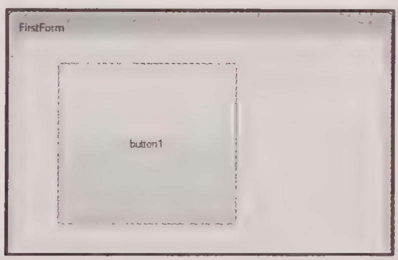


Fig.C#-5.17

In Fig.C#-5.17, when you increase the size of the **FirstForm** form, the size of **button1** also increases and vice versa.

Enabling and Disabling Controls

You can set various properties of controls on a form. For instance, you can enable or disable controls by setting their properties using the **Properties** window. For disabling a control, simply set the **Enabled** property to **False** in the **Properties** window of the control. Similarly, you can enable a control by setting the **Enabled** property to **True**. By default, the **Enabled** property of a control is set to **True**.

Let's perform the following steps to hide the **Button** control in the **SampleApplication** application:

1. *Select* the **Button** control on the **FirstForm** form (Fig.C#-5.18).
2. *Set* the **Enabled** property of the **Button** control to **False** in the **Properties** window, as shown in Fig.C#-5.18:

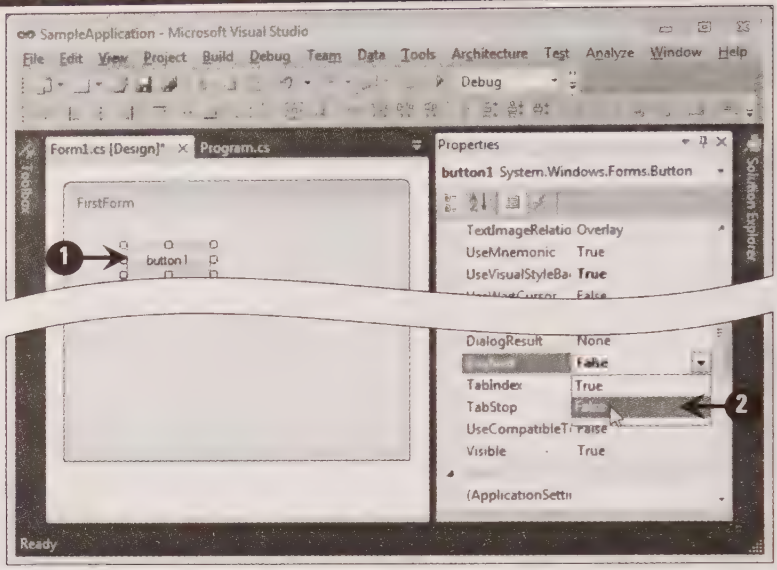


Fig.C#-5.18

3. *Press* the **F5** key to run the **SampleApplication** application. The output appears, as shown in Fig.C#-5.19:

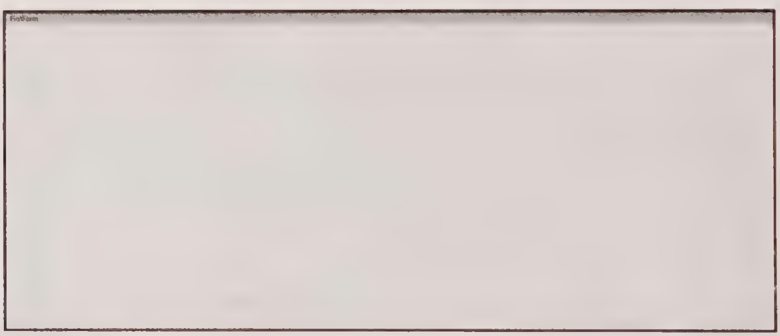


Fig.C#-5.19

In Fig.C#-5.19, the **button1** Button control appears disabled.

Specifying the Tab Order of Controls

Controls on a form can be made accessible in an appropriate sequence by setting their **TabIndex** property. This property allows you to navigate all the controls in a logical manner by pressing the **Tab** key.

Let's perform the following steps to specify the tab order of controls:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **TabOrder** in the **Name** text box and specify an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **TabOrder** application is created with the specified name and location.
4. *Drag* two **Button** controls and two **TextBox** controls from **Toolbox** and *drop* these controls on the **Form1** form (Fig.C#-5.20).
5. *Set* the **Text** property of **button1** to **Add Name** and **button2** to **Add Age**, as shown in Fig.C#-5.20:

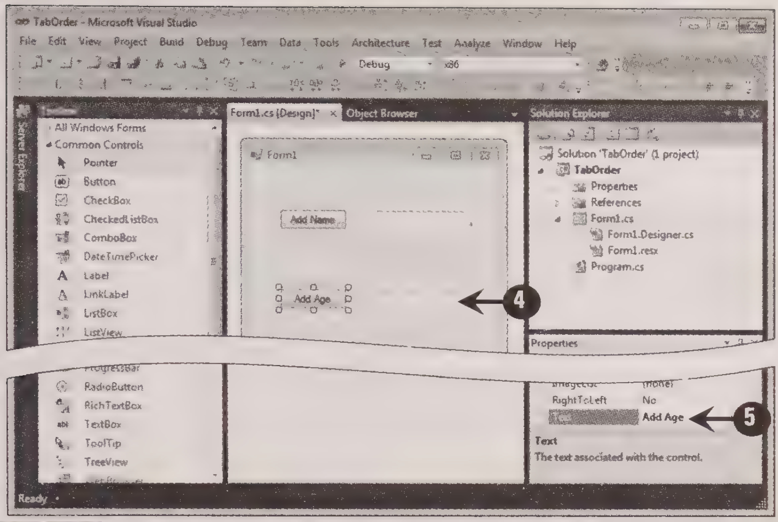


Fig.C#-5.20

6 *Select* **button1** and *set* its **TabIndex** property to **0** (zero), as shown in Fig.C#-5.21:

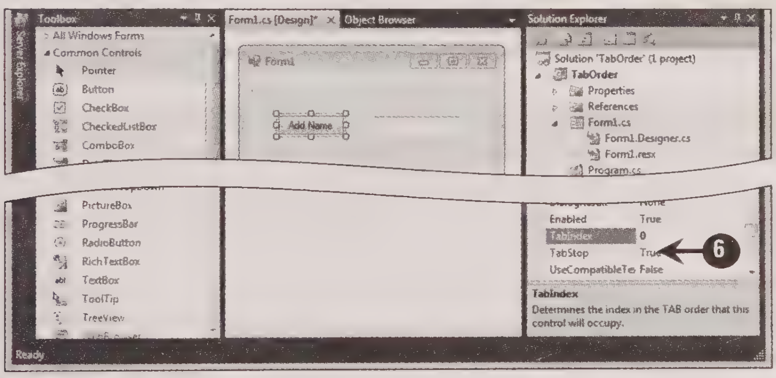


Fig.C#-5.21

7 *Repeat* step 6 for all other controls in the **Form1** form by assigning each control a **TabIndex** property value that is one more than that of its previous control.

Note

While setting the tab order of a control, make sure that its TabStop property is set to True. If this property is False, the user cannot reach the control using the Tab key.

8 *Press* the **F5** key to run the **TabOrder** application. The focus is on the **Add Name** button, as shown in Fig.C#-5.22.

Fig.C#-5.22

9. *Press* the **Tab** key. The output appears, as shown in Fig.C#-5.23:

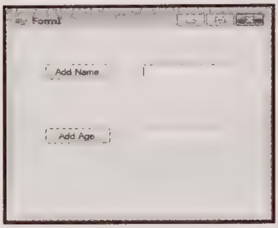


Fig.C#-5.23

In Fig.C#-5.23, the focus is set to the second control, the control having the **TabIndex** value **1**. In our case, it is **textbox1**.

Handling Common Events for Windows Forms Applications

C# is an event driven language, a language in which the flow of the program is controlled by user actions, such as pressing a key or clicking a control. You can write code to handle an event in the code designer.

There are mainly two types of events:

- **Mouse events:** Refers to events generated by supplying the input through a mouse
- **Keyboard events:** Refers to events generated by supplying the input through keyboard

Let's first learn to handle mouse events.

Handling Mouse Events

The most common mouse event associated with controls in C# is the **Click** event.

Let's learn to handle the **Click** event of a **Button** control by performing the following steps:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **MouseEvent** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **MouseEvent** application is created with the specified name and location.
4. *Drag* a **Button** control and **TextBox** control from **Toolbox** and *drop* these controls on the **Form1** form (Fig.C#-5.24).
5. *Set* the **Text** property of **button1** to **Click** in the **Properties** window, as shown in Fig.C#-5.24:

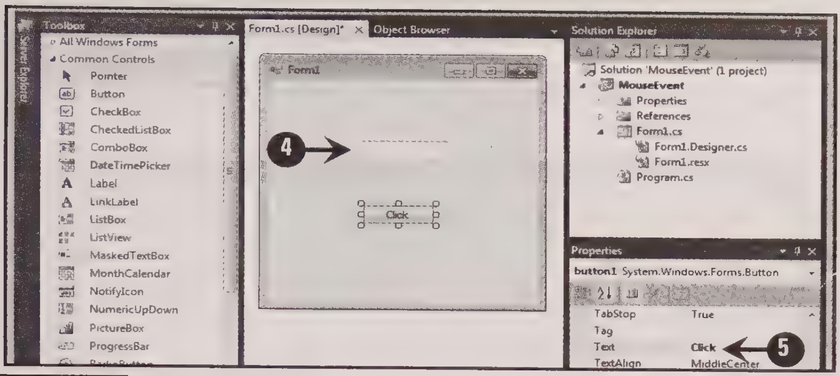


Fig.C#-5.24

6. *Double-click* the **Button** control and add the highlighted code given in Listing 5.3 in Code Editor, to handle the **Click** event of the **Button** control:

Listing 5.3: Showing the Code to Handle the Click Event of the Button Control

```
private void button1_Click(object sender, EventArgs e)
{
        textBox1.Text = "Welcome to C# 2010";
}
```

7. *Press* the **F5** key to run the **MouseEvent** application.

   The output after clicking the **Click** button is shown in Fig.C#-5.25:

Fig.C#-5.25

In Fig.C#-5.25, the text **Welcome to C# 2010** is displayed in text box.

Now, let's learn to handle keyboard events.

Handling Keyboard Events

In C#, along with mouse events, you can also generate keyboard events. The different keyboard events in Windows Forms application are **KeyDown**, **KeyPress**, and **KeyUp**.

Let's learn to handle the **KeyPress** event of a **TextBox** control by performing the following steps:

1. *Repeat* steps 1-3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **KeyboardEvent** in the **Name** text box and specify an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **KeyboardEvent** application is created with the specified name and location.
4. *Drag* a **TextBox** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.26).
5. *Select* the text box control you just added and open the **Properties** window to handle the **KeyPress** event of the **TextBox** control (Fig.C#-5.26).

6 *Click* the **Events** button in the **Properties** window, all the events related to text box are displayed in the **Properties** window (Fig.C#-5.26).

7 *Double-click* beside the **KeyPress** event to generate the **textBox1_KeyPress** event handler, as shown in Fig.C#-5.26:

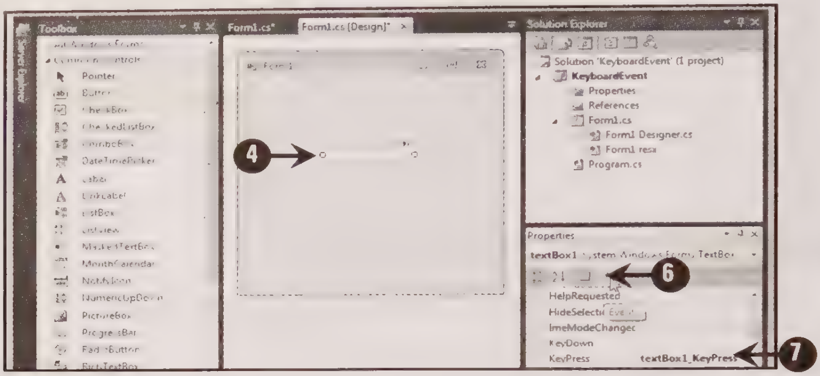


Fig.C#-5.26

8 *Add* the highlighted code in Code Editor, as given in Listing 5.4:

Listing 5.4: Showing the Code to Handle the KeyPress Event of the TextBox Control

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
  {
        textBox1.BackColor = Color.Red;
        textBox1.ForeColor = Color.White;
  }
```

9 *Press* the **F5** key to run the **KeyboardEvent** application.

When you type some text in the **TextBox** control, the back color of the **TextBox** control changes to red and the text color changes to white, as shown in Fig.C#-5.27:

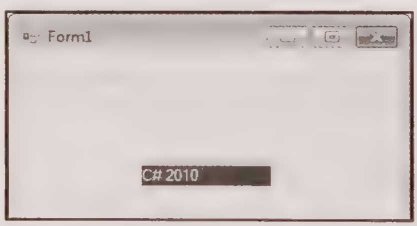


Fig.C#-5.27

In this section, you learned about handling common events of a Windows Forms application. Now, let's learn to add controls to a form and perform certain operations on them.

Working with Windows Forms Controls

Controls are components that can be placed on the form and used to display some information, accept some user input, and perform text editing. In addition, Windows controls are the basic controls that are used for creating applications in C# 2010. A control enhances user interaction with Windows Forms application. These controls are the common controls that are available in **Toolbox** of C# 2010. The common controls in Windows Forms application are as follows:

- The Button control
- The Label control
- The TextBox control
- The RichTextBox control
- The RadioButton control
- The CheckBox control
- The ListBox control
- The ComboBox control
- The ListView control
- The PictureBox control
- The ProgressBar control
- The TabControl control
- The GroupBox control
- The Timer control

Now, let's learn to use the **Button** control.

Using the Button Control

The Button control is one of the most basic Windows Forms controls. Every Windows Forms application has at least one Button control associated with it. The Button control allows you to generate the Click event. A user can perform some action at the runtime by handling the Click event of the Button control.

Let's perform the following steps to use the Button control in a Windows Forms application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **WindowsControls** in the **Name** text box and specify an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **WindowsControls** application is created with the specified name and location.
4. *Drag* a **Button** control from **Toolbox** and *drop* it on the **Form1** form.
5. *Set* the **Text** property of the **Button** control to **Click** in the **Properties** window, as shown in Fig.C#-5.28:

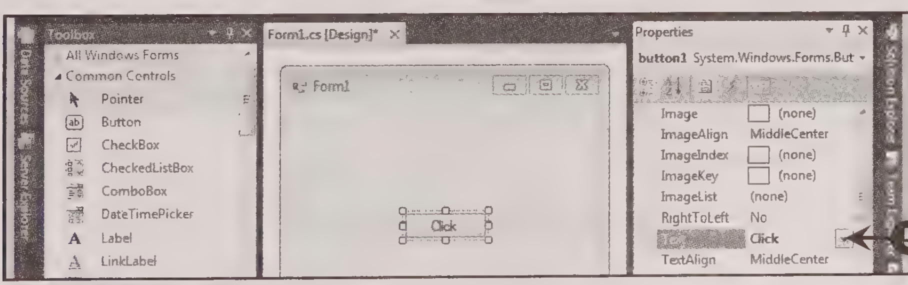


Fig.C#-5.28

In Fig.C#-5.28, the text of the **Button** control changes to **Click**.

Using the Label Control

The Label control is a control that is used to display text. You can use it for displaying static text that you do not want to be edited by a user. It can also be used to display dynamic text (text that changes after the occurrence of

an event in an application). You need to set the properties of the **Label** control to customize the appearance of a label.

Let's perform the following steps to use the **Label** control in the **WindowsControls** application:

1. *Drag* a **Label** control from **Toolbox** and *drop* it anywhere on the **Form1** form (Fig.C#-5.29)
2. *Set* the **Text** property of the **Label** control to **Name** from the **Properties** window, as shown in Fig.C#-5.29:

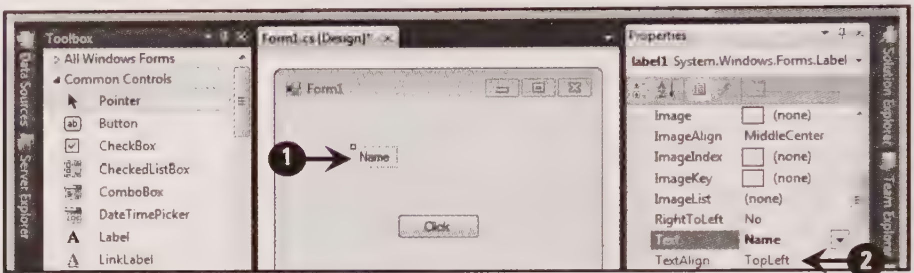


Fig.C#-5.29

In Fig.C#-5.29, the text of the **Label** control changes to **Name**.

Using the TextBox Control

The TextBox control is a Windows Forms control that allows you to enter text on a form at runtime. This control is usually used when the user requires simple text area where one or few lines of text can be displayed. By default, a TextBox control accepts only a single line of text. However, by setting various properties of the TextBox control, you can customize it to accept multiple lines of text, add scroll bars to the control, and disable text editing in the control.

Let's perform the following steps to use the **TextBox** control in the **WindowsControls** application:

1. *Drag* a **TextBox** control from **Toolbox** and *drop* it on the **Form1** form.
2. *Double-click* the **Click** button control added in the **Using the Button Control** section of this chapter and add the highlighted code given in Listing 5.5:

Listing 5.5: Adding code on the button1_Click Event of the Button Control

```
private void button1_Click(object sender, EventArgs e)
  {
        textBox1.ForeColor = Color.Blue;
        textBox1.Text = "Hello C#";
        label1.BackColor = Color.Red;
}
```

In Listing 5.5, the **ForeColor** property of a control is used to define the fore color of the **TextBox** control. In our case, we have set it to blue color. The **Text** property of a control is used to define text in the **TextBox** control. The **BackColor** property of a control is used to define the back color of the **Label** control. In our case, we have set it to red color.

3. *Press* the **F5** key to run the **WindowsControls** application. The output is shown in Fig.C#-5.30:

4 *Drag* three **RadioButton** controls and a **Label** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.35).

5 *Set* the **Text** property of **radioButton1** to **Red** in the **Properties** window, as shown in Fig.C#-5.35:

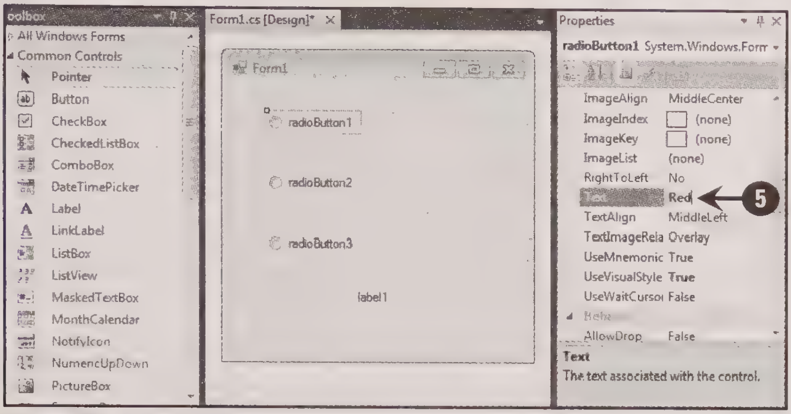


Fig.C#-5.35

6 *Change* the **Text** property of **radioButton2** to **Blue**, **radioButton3** to **Green**, and **label1** to **Welcome** from the **Properties** window.

7 *Double-click* the **Red** radio button and add the highlighted code shown in Listing 5.7 on the **CheckedChanged** event of the **Red** radio button:

Listing 5.7: Adding Code on the CheckedChanged Event of the Red Radio Button

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
   label1.ForeColor = Color.Red;
}
```

In Listing 5.7, when you select the **Red** radio button, the fore color of the **label1** label changes to red by using its **ForeColor** property.

8 *Add* the highlighted code shown in Listing 5.8 on the **CheckedChanged** event of the **Blue** radio button:

Listing 5.8: Adding Code on the CheckedChanged Event of the Blue Radio Button

```
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
   label1.ForeColor = Color.Blue;
}
```

In Listing 5.8, when you select the **Blue** radio button, the font color of the label changes to blue by using the **ForeColor** property of the **label1** control.

9 *Add* the highlighted code shown in Listing 5.9 on the **CheckedChanged** event of the **Green** radio button:

Listing 5.9: Adding Code on the CheckedChanged Event of the Green Radio Button

```
private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
   label1.ForeColor = Color.Green;
}
```

In Listing 5.9, when you select the **Green** radio button, the font color of the label changes to blue by using the **ForeColor** property of the **label1** control.

10 *Press* the **F5** key to run the **ControlApplication** application and *select* any of the three radio buttons. In our case, we selected the **Blue** radio button. The output appears, as shown in Fig.C#-5.36:

6. *Double-click* the **Button** control and add the highlighted code shown in Listing 5.6 in the **Form1.cs** file:

Listing 5.6: Adding Code on the Click Event of the Click Button Control

```
private void button1_Click(object sender, EventArgs e)
{
    RichTextBox richTextBox1 = new RichTextBox();
    richTextBox1.Dock = DockStyle.Top;
    richTextBox1.Height = 150;
    richTextBox1.Width = 50;
    this.Controls.Add(richTextBox1);
    richTextBox1.Text = "Hello! Welcome to the world of C# Programming. This is the new
    version of C#.";
}
```

In Listing 5.6, the code adds a **RichTextBox** control, **richTextBox1**, at the **Click** event of **button1**. The **Height** and **Width** properties of **richTextBox1** are set to **150** and **50** respectively. In addition, the **Text** property of **richTextBox1** contains the message, **Hello! Welcome to the world of C# Programming. This is the new version of C#.**

7. *Press* the **F5** key to run the **RichTextBoxControl** application. The output appears (Fig.C#-5.33).
8. *Click* the **Click** button, as shown in Fig.C#-5.33:

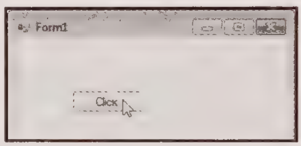


Fig.C#-5.33

The output appears, as shown in Fig.C#-5.34:

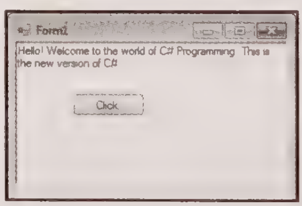


Fig.C#-5.34

In Fig.C#-5.34, a **RichTextBox** control with some text appears in the **Form1** form.

Using the RadioButton Control

A RadioButton control, also known as option button, is used to select one option from a set of options. This control always works in groups. This means that whenever you select one RadioButton control from a group of RadioButton controls, the other RadioButton controls in the group are automatically deselected. A RadioButton control can display text, an image, or both.

Let's perform the following steps to use the **RadioButton** control in a Windows Forms application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section.
2. *Enter* **ControlApplication** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **ControlApplication** Windows Forms application is created with the specified name and location.

4. *Drag* three **RadioButton** controls and a **Label** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.35).

5. *Set* the **Text** property of **radioButton1** to **Red** in the **Properties** window, as shown in Fig.C#-5.35:

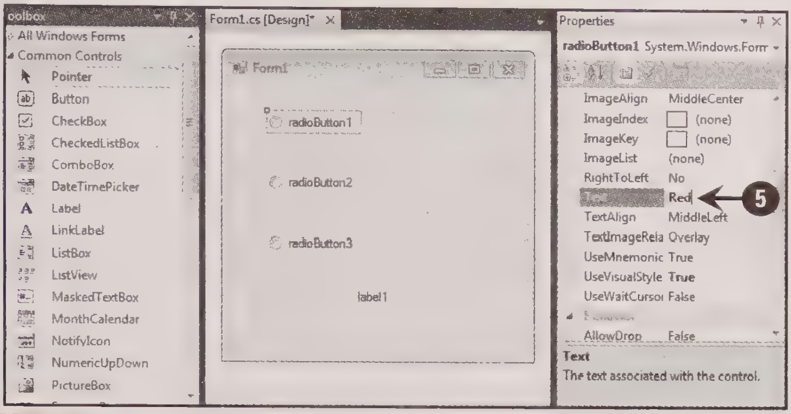


Fig.C#-5.35

6. *Change* the **Text** property of **radioButton2** to **Blue**, **radioButton3** to **Green**, and **label1** to **Welcome** from the **Properties** window.

7. *Double-click* the **Red** radio button and add the highlighted code shown in Listing 5.7 on the **CheckedChanged** event of the **Red** radio button:

Listing 5.7: Adding Code on the CheckedChanged Event of the Red Radio Button

```
private void radioButton1_CheckedChanged(object sender, EventArgs e)
{
   label1.ForeColor = Color.Red;
}
```

In Listing 5.7, when you select the **Red** radio button, the fore color of the **label1** label changes to red by using its **ForeColor** property.

8. *Add* the highlighted code shown in Listing 5.8 on the **CheckedChanged** event of the **Blue** radio button:

Listing 5.8: Adding Code on the CheckedChanged Event of the Blue Radio Button

```
private void radioButton2_CheckedChanged(object sender, EventArgs e)
{
   label1.ForeColor = Color.Blue;
}
```

In Listing 5.8, when you select the **Blue** radio button, the font color of the label changes to blue by using the **ForeColor** property of the **label1** control.

9. *Add* the highlighted code shown in Listing 5.9 on the **CheckedChanged** event of the **Green** radio button:

Listing 5.9: Adding Code on the CheckedChanged Event of the Green Radio Button

```
private void radioButton3_CheckedChanged(object sender, EventArgs e)
{
   label1.ForeColor = Color.Green;
}
```

In Listing 5.9, when you select the **Green** radio button, the font color of the label changes to blue by using the **ForeColor** property of the **label1** control.

10. *Press* the **F5** key to run the **ControlApplication** application and *select* any of the three radio buttons. In our case, we selected the **Blue** radio button. The output appears, as shown in Fig.C#-5.36:

Fig.C#-5.36

In Fig.C#-5.36, the fore color of the label is changed to blue when the **Blue radio button** is selected.

Using the CheckBox Control

A CheckBox control accepts a value of either true or false. You need to just click the CheckBox control to select it. When the CheckBox control is selected a check mark appears; click the control again to deselect it. When you select the CheckBox control, it holds the value, true and when you deselect the CheckBox control, it holds the value, false. A CheckBox control can display an image or corresponding text associated with it. It can also display both at the same time.

Let's perform the following steps to use the **CheckBox** control in the **ControlApplication** application:

1. *Drag* two **CheckBox** controls from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.37).
2. *Set* the **Text** property of **checkBox1** to **Bold** in the **Properties** window, as shown in Fig.C#-5.37:

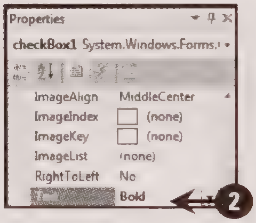


Fig.C#-5.37

3. *Change* the **Text** property of **checkBox2** to **Italic** in the **Properties** window.
4. *Double-click* the check box, **Bold** and *add* the highlighted code shown in Listing 5.10 on the **CheckedChanged** event of **checkBox1**:

Listing 5.10: Adding Code on the CheckedChanged Event of checkBox1

```
private void checkBox1_CheckedChanged(object sender, EventArgs e)
{
   label1.Font = new Font(label1.Font.Name, label1.Font.Size, label1.Font.Style |
   FontStyle.Bold);
}
```

In Listing 5.10, the code changes the font of the text displayed on the **label1** label to bold when you select the **Bold** check box.

5. *Add* the highlighted code shown in Listing 5.11 on the **CheckedChanged** event of **checkBox2**:

Listing 5.11: Adding Code on the CheckedChanged Event of checkBox2

```
private void checkBox2_CheckedChanged(object sender, EventArgs e)
{
```

```
    label1.Font = new Font(label1.Font.Name, label1.Font.Size, label1.Font.Style |
    FontStyle.Italic);
}
```

The code given in Listing 5.11 changes the text font of label to italic when you select the **Italic** check box.

6. *Press* the **F5** key to run the **ControlApplication** application and select the **Red** radio button and **Bold** check box, the output appears, as shown in Fig.C#-5.38:

Fig.C#-5.38

In Fig.C#-5.38, the text in **label1** appears in red color and bold.

Using the ListBox Control

ListBox control is a standard Windows control that is used to display text as a list. Text can be displayed as a sorted or an unsorted list. The ListBox control recognizes text as a collection of items. You can add text as an item into this collection to display it on a ListBox control. Similarly, you can remove an item from the list if you do not want to display it on the ListBox control. Each item in a ListBox control is recognized by its position in the list, which is known as its index.

Let's perform the following steps to use the **ListBox** control in a Windows Forms application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **ListAndComboBox** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **ListAndComboBox** application is created with the specified name and location.
4. *Drag* a **ListBox** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.39).
5. *Click* the ellipsis button of the **Items** property in the **Properties** window of the **ListBox** control, as shown in Fig.C#-5.39:

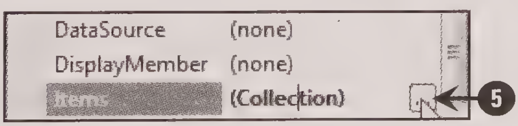


Fig.C#-5.39

The **String Collection Editor** dialog box opens (Fig.C#-5.40).

6. *Enter* the required items in the **String Collection Editor** dialog box (Fig.C#-5.40).
7. *Click* the **OK** button, as shown in Fig.C#-5.40:

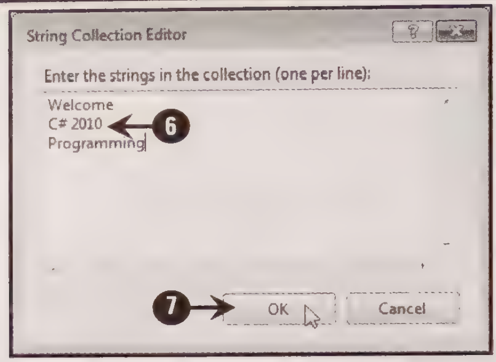


Fig.C#-5.40

The selected items are added to the **ListBox** control, as shown in Fig.C#-5.41:

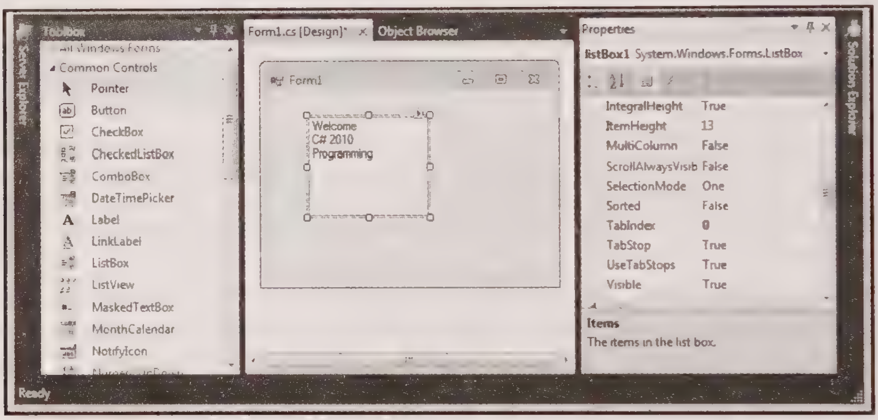


Fig.C#-5.41

In Fig.C#-5.41, the **ListBox** control that is added to **Form1** is displayed.

Using the ComboBox Control

A ComboBox control is a .NET control that is widely used to select an option from a list. The ComboBox control is used to display data in a drop-down list. When the user selects an item from the combo box, the list contained in the ComboBox control automatically collapses. A user can choose only a single item from the expandable list of items. You can add or remove an item from this list. Each item in a ComboBox control is recognized by its position in the list, which is known as an index.

Let's perform the following steps to use the **ComboBox** control in the **ListAndComboBox** application:

1. *Add* a **ComboBox** control to the **ListAndComboBox** application from **Toolbox** (Fig.C#-5.42).
2. *Click* the ellipsis button of the **Items** property in the **Properties** window of the **ComboBox** control, as shown in Fig.C#-5.42:

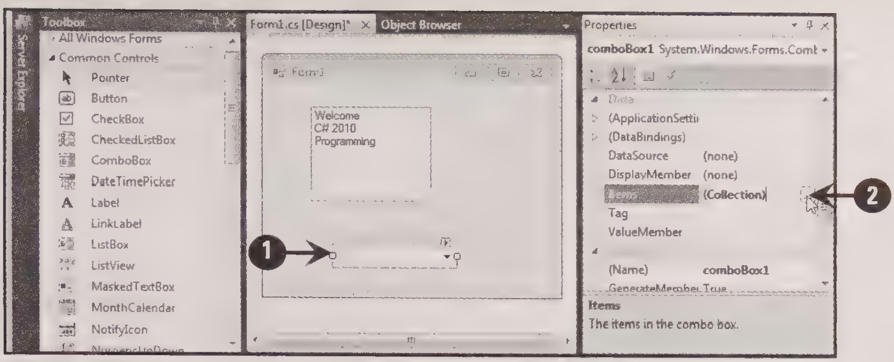


Fig.C#-5.42

The **String Collection Editor** dialog box appears.

3. *Enter* the required items in the **String Collection Editor** dialog box.
4. *Click* the **OK** button.

   This adds the items that you entered in the **String Collection Editor** dialog box to the **ComboBox** control.
5. *Double-click* the **ComboBox** control and *add* the highlighted code given in Listing 5.12 in the **Form1.cs** file:

Listing 5.12: Adding Code on the comboBox1_SelectedIndexChanged Event

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
{
  if (comboBox1.SelectedItem.ToString() == "It")
  {
       listBox1.Items.Add(comboBox1.SelectedItem.ToString());
  }
  if (comboBox1.SelectedItem.ToString() == "is a")
  {
       listBox1.Items.Add(comboBox1.SelectedItem.ToString());
  }
  if (comboBox1.SelectedItem.ToString() == "ComboBox")
  {
       listBox1.Items.Add(comboBox1.SelectedItem.ToString());
  }
}
```

6. *Press* the **F5** key to run the application.

   The output appears, as shown in Fig.C#-5.43:

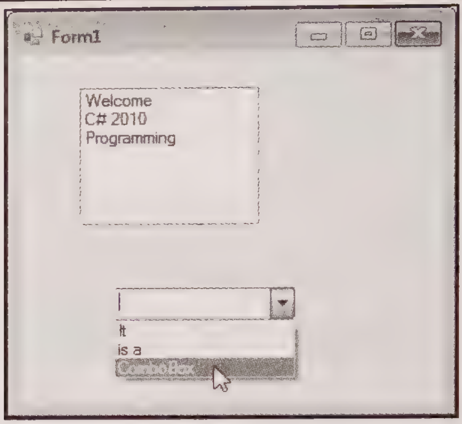


Fig.C#-5.43

In Fig.C#-5.43, as you select a value from the **ComboBox** control, the value is added to the **ListBox** control.

Using the ListView Control

The ListView controls are used to display lists of items, just as tree views are used to display node hierarchies that are similar to the folder hierarchy on the hard disk of a computer.

Let's perform the following steps to use the **ListView** control in a Windows Forms application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **ListView** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the OK button. The **ListView** application is created with the specified name and location.
4. *Drag* a **ListView** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.44).
5. *Click* the ellipsis button of the **Items** property in the **Properties** window, as shown in Fig.C#-5.44:

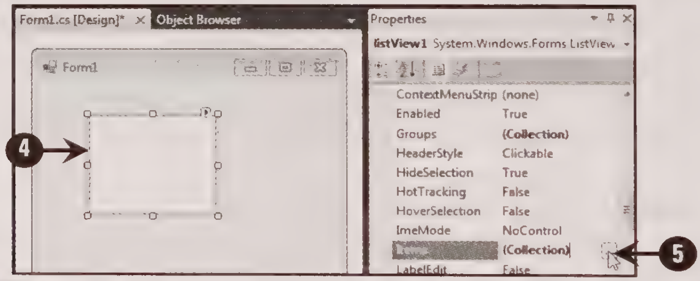


Fig.C#-5.44

The **ListViewItem Collection Editor** dialog box appears (Fig.C#-5.45).

6. *Click* the **Add** button, as shown in Fig.C#-5.45:

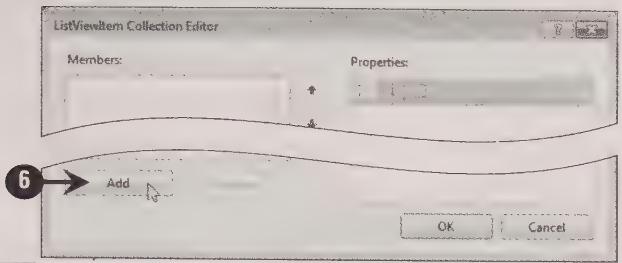


Fig.C#-5.45

This adds the **ListViewItem{}** collection in the **Members** pane of the **ListViewItem Collection Editor** dialog box (Fig.C#-5.46).

7 *Set* the **Text** property to **FirstItem** in the **ListViewItem{}** properties pane (Fig.C#-5.46).

8 *Click* the **OK** button, as shown in Fig.C#-5.46:

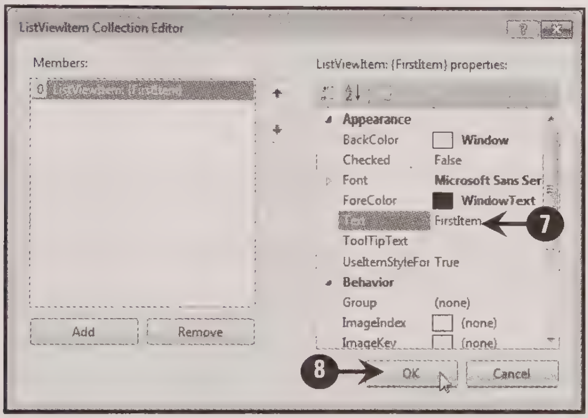


Fig.C#-5.46

The **FirstItem** item is added to the **ListView** control, as shown in Fig.C#-5.47:

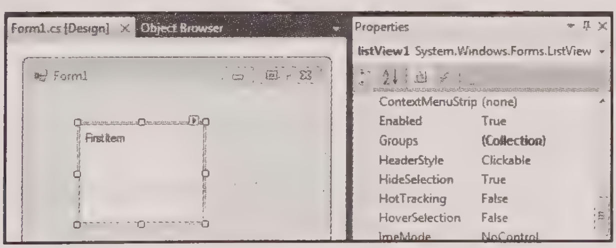


Fig.C#-5.47

In Fig.C#-5.47, the **ListView** control is displayed in the **Form1** form.

Using the TabControl Control

The TabControl control in C# displays multiple tabs and is designed to conserve space. Each tab contains multiple items that share the same space on the TabControl control.

Let's perform the following steps to use the **TabControl** control in a Windows Forms application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **TabControl** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **TabControl** application is created with the specified name and location.
4. *Drag* a **TabControl** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.48).
5. *Click* the ellipsis button of the **TabPages** property in the **Properties** window, as shown in Fig.C#-5.48:

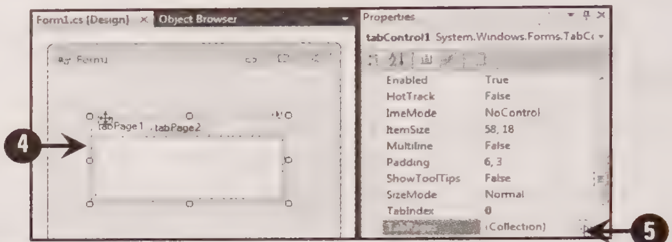


Fig.C#-5.48

The **TabPage Collection Editor** dialog box appears (Fig.C#-5.49).

6. *Change* the **Text** property to **FirstTab** in the **tabpage1 properties** pane (Fig.C#-5.49).
7. *Click* the **OK** button, as shown in Fig.C#-5.49:

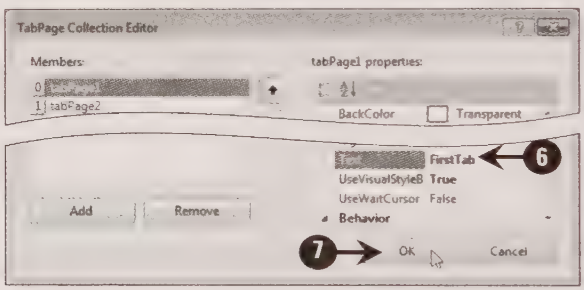


Fig.C#-5.49

This changes the text of first tab of the **TabControl** control to **FirstTab**.

Next, let's discuss about the **GroupBox** control.

Using the GroupBox Control

A GroupBox control is used to group various controls in a logical order. Group boxes can have a caption. Mainly, you can use group boxes to sub divide a form on functional basis.

Let's perform the following steps to use the **GroupBox** control in a Windows Forms application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **GroupingControls** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **GroupingControls** application is created with the specified name and location.
4. *Drag* a **GroupBox** control from **Toolbox** and *drop* it on the **Form1** form (Fig.C#-5.50).

Note

You can find the GroupBox control in the Containers tab of Toolbox.

5. *Change* the **Text** property of the **GroupBox** control to **Grouping** from the **Properties** window, as shown in Fig.C#-5.50:

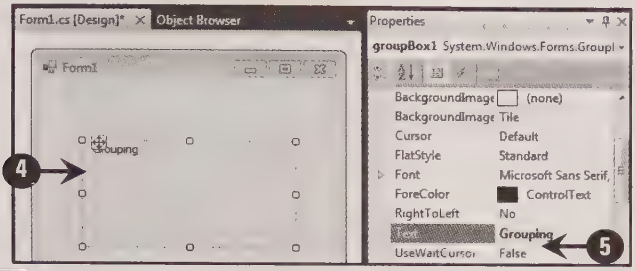


Fig.C#-5.50

The text of the **GroupBox** control changes to **Grouping**.

Using the PictureBox Control

A PictureBox control is a Windows control that is used to display images in a Windows Forms application. We can add a picture in a form with the help of PictureBox control. A picture can also be edited in the PictureBox control. For instance, you can stretch the image you have added in the **PictureBox** control.

Let's perform the following steps to use the **PictureBox** control in the **GroupingControls** application:

1. *Drag* a **PictureBox** control from **Toolbox** and *drop* it into **Form1**, in the **GroupBox** control (Fig.C#-5.51).
2. *Click* the ellipsis button of the **Image** property in the **Properties** window, as shown in Fig.C#-5.51:

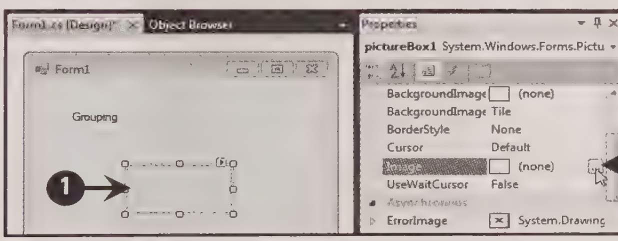


Fig.C#-5.51

The **Select Resource** dialog box appears.

3. *Select* the **Local resource** radio button.
4. *Click* the **Import** button.

The **Open** dialog box appears.

5. *Select* the image that you want to add in the **PictureBox** control.
6. *Click* the **Open** button.

The selected image is loaded in the **Select Resource** dialog box (Fig.C#-5.52).

7. *Click* the **OK** button, as shown in Fig.C#-5.52:

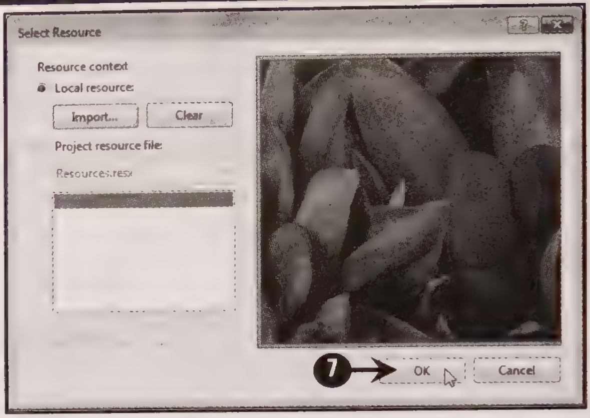


Fig.C#-5.52

The image is added in the **PictureBox** control, as shown in Fig.C#-5.53:

Fig.C#-5.53

In Fig.C#-5.53, the selected picture is displayed in the **PictureBox** control.

Using the ProgressBar Control

The ProgressBar control is used to indicate the progress of a process. It shows a bar that fills itself from left to right as the operation progresses. The Minimum and Maximum properties indicate the range of values representing the progress of a task. Usually, Minimum property is set to zero and Maximum property is set to a value that indicates the completion of a task.

Let's perform the following steps to use the **ProgressBar** control in the **ProgressBarControl** application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.

2. *Enter* **ProgressBarControl** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **ProgressBarControl** application is created with the specified name and location.
4. *Drag* a **ProgressBar** control, a **Button** control and a **Timer** control from **Toolbox** and *drop* on the **Form1** form (Fig.C#-5.54).

Note

You learn about the Timer control in detail in the Using the Timer Control section.

5. *Change* the **Text** property of the **Button** control to **Click Me** from the **Properties** window, as shown in Fig.C#-5.54:

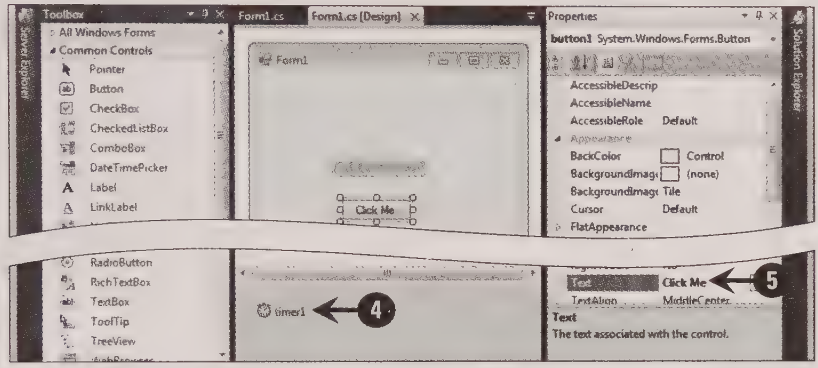


Fig.C#-5.54

6. *Double-click* the **button1** control and *add* the highlighted code given in Listing 5.13 on the **Click** event of the **button1** button:

Listing 5.13: Adding Code on the button1_Click Event

```
private void button1_Click(object sender, EventArgs e)
{
   timer1.Enabled = true;
}
```

7. *Add* the highlighted code given in Listing 5.14 on the **Tick** event of **timer1**:

Listing 5.14: Adding Code on the timer1_Tick Event

```
private void timer1_Tick(object sender, EventArgs e)
{
   progressBar1.Value += 1;
   if (progressBar1.Value == progressBar1.Maximum)
   {
        timer1.Enabled = false;
MessageBox.Show("Timer Has Been Disabled", "Timer Disabled", MessageBoxButtons.OKCancel,
MessageBoxIcon.Exclamation);
   }
}
```

8. *Press* the **F5** key to run the **ProgressBarControl** application.

The output appears, as shown in Fig.C#-5.55:

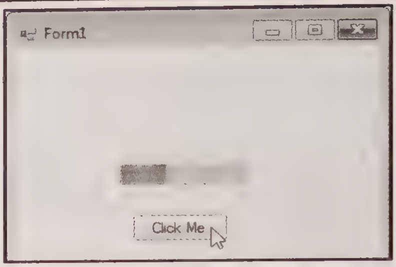


Fig.C#-5.55

When the **Value** property of the progress bar becomes equal to its maximum value, the timer is disabled and the output appears, as shown in Fig.C#-5.56:

Fig.C#-5.56

As shown in Fig.C#-5.56 a message box appears, displaying a message that timer is disabled.

Using the Timer Control

Timer controls are used to generate periodic events. These controls are called components and they do not appear in a window at run time. At design time, they appear in the component tray below the form in which they are added.

Let's perform the following steps to use the **Timer** control in the **TimerControl** application:

1. *Repeat* steps 1 to 3 of the **Changing the Title of a Form** section of this chapter.
2. *Enter* **TimerControl** in the **Name** text box and *specify* an appropriate location in the **Location** combo box.
3. *Click* the **OK** button. The **TimerControl** application is created with the specified name and location.
4. *Drag* a **Timer**, two **Button** controls and a **ListBox** control from **Toolbox** and *drop* on **Form1** (Fig.C#-5.57).
5. *Change* the **Text** property of the **Button1** control to **Click To Start Timer** and the **Text** property of **Button2** control to **Click To Stop Timer** from the **Properties** window, as shown in Fig.C#-5.57:

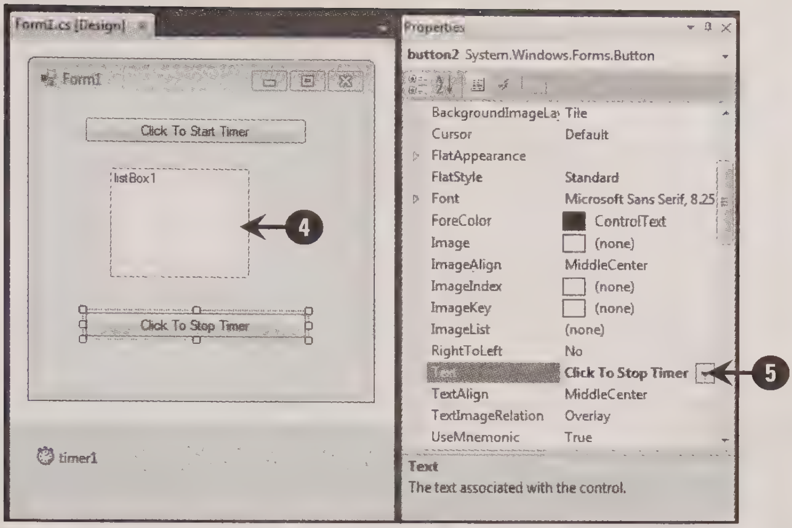


Fig.C#-5.57

6 *Double-click* the **button1** control and *add* the highlighted code given in Listing 5.15 on the **Click** event of **button1**:

Listing 5.15: Adding Code on the button1_Click Event

```
private void button1_Click(object sender, EventArgs e)
{
  timer1.Enabled = true;
}
```

7 *Add* the highlighted code given in Listing 5.16 on the **Tick** event of **timer1**:

Listing 5.16: Adding Code on **timer1_Tick** Event

```
private void timer1_Tick(object sender, EventArgs e)
{
  listBox1.Items.Add("Timer Enabled");
  listBox1.Items.Add(DateTime.Now.ToString());
}
```

8 *Double-click* the **button2** control, and *add* the highlighted code given in Listing 5.17 on the **Click** event of **button2**:

Listing 5.17: Adding Code on button2_Click Event

```
private void button2_Click(object sender, EventArgs e)
{
  timer1.Enabled = false;
  listBox1.Items.Add("Timer Disabled");
}
```

9 *Press* the **F5** key to run the **TimerControl** application. The output appears (Fig.C#-5.58).

10 *Click* the **Click To Start Timer** button, as shown in Fig.C#-5.58:

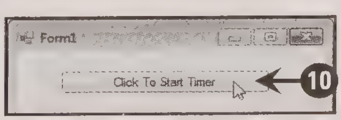


Fig.C#-5.58

The **Timer** control starts and the current date and time are displayed on the **ListBox** control.

11 *Click* the **Click To Stop Timer** button to disable the **Timer** control, as shown in Fig.C#-5.59:

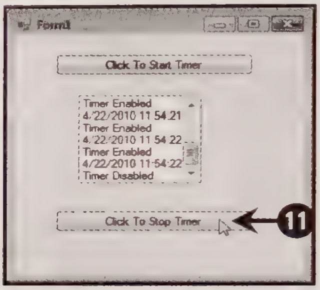


Fig.C#-5.59

With this we come to the end of the chapter, let's summarize the key points of the chapter.

Summary

In this chapter, you have learned about:

- Performing common operations on a form
- Performing event handling in a Windows Forms application
- Working with various controls used in a Windows Forms application

Chapter 6

Working with Windows Forms Menus, Toolbars, and Dialog Controls

In this Chapter:

- Creating Toolbars, Menus, and Status Bar in C# 2010
- Working with Dialog Boxes
-
- Creating Toolbars, Menus, and Status Bar in C# 2010
- Working with Dialog Boxes

At times, you may come across situations where you need to provide different options to users so that the users can perform several actions on a particular entity, such as creating, opening, saving, or printing a file. You can use check boxes to display all these options to the user. However, if there is large number of options it would not be possible to display all of them on a form as they take a lot of space. Moreover, it would be rather difficult for the user to locate the right option on the form. In such cases, the use of toolbars and menus offers an easy, convenient, and compact means of providing various options and tasks. Toolbars and menus are used to group related options. For example, you can group operations related to a file, such as creating, opening, and saving a file, under the File menu. Toolbars act as shortcuts to menus and are used to perform some common functions. In addition to toolbars and menus, C# 2010 also provides you various controls to work with dialog boxes.

In this chapter, you learn how to create toolbars, menus, and status bar by using the **ToolStrip**, **MenuStrip**, and **StatusStrip** controls, respectively. You also learn how to work with some common dialog boxes, such as, the **Open**, **Save As**, and **Print** dialog boxes.

Let's start by learning about toolbars, menus, and status bar.

Creating Toolbars, Menus, and Status Bar in C# 2010

As you know, toolbars and menus present a compact and convenient way of presenting different options. Toolbars provide commonly used menu items. In Windows Forms applications, you can create toolbars and menus by using the **ToolStrip** and **MenuStrip** controls, respectively. You can also display information about these controls and your application by using a status strip, provided by the **StatusStrip** control.

Let's discuss these controls in detail, starting with the **ToolStrip** control.

Using the ToolStrip Control

The **ToolStrip** control is an improved version of the **ToolBar** control. It offers a strip of tools that correspond to menu options and controls, such as buttons, images, drop-down lists, and text boxes, which allow you to develop Windows Forms applications with rich yet compact user interface (UI). The **ToolStrip** control offers various commonly-used features and options to the users, such as layout options. It also allows the users to execute frequently performed tasks, such as saving files.

The **ToolStrip** control is an instance of the **System.Windows.Forms.ToolStrip** class. This class provides various properties, methods, and events that allow you to work with the **ToolStrip** control. Table 6.1 lists noteworthy properties of the **ToolStrip** class:

Table 6.1: Noteworthy Properties of the ToolStrip Class

Property	Description
Anchor	Retrieves or sets the edges of the container to which the ToolStrip control is bound
AutoSize	Retrieves or sets a value that indicates whether the ToolStrip control is automatically resized to display its entire content
Dock	Retrieves or sets the borders of the ToolStrip control that is docked to its parent control
GripDisplayStyle	Retrieves the orientation of the move handle of the ToolStrip control
GripStyle	Retrieves or sets a value that indicates whether the ToolStrip control move handle is visible or hidden
ImageList	Retrieves or sets the image list that contains the images displayed on a tool strip item
IsDropDown	Retrieves a value that indicates whether the ToolStrip control is a ToolStripDownControl
Items	Retrieves all the items of the ToolStrip control
LayoutStyle	Retrieves or sets a value that indicates the way the ToolStrip control places its tool strip items inside a form

Table 6.1: Noteworthy Properties of the ToolStrip Class	
Property	**Description**
Orientation	Retrieves the orientation of the ToolStripPanel object
ShowItemToolTips	Retrieves or sets a value indicating whether ToolTips for the tool strip items are displayed
Stretch	Retrieves or sets a value that indicates whether the ToolStrip control stretches from end to end in the ToolStripContainer class
TabStop	Retrieves or sets a value indicating whether a user can give focus to a tool strip item using the TAB key
TextDirection	Retrieves or sets the direction in which the text on the ToolStrip control appears

Table 6.2 lists noteworthy methods of the **ToolStrip** class:

Table 6.2: Noteworthy Methods of the ToolStrip Class	
Method	**Description**
`GetItemAt`	Retrieves the tool strip item at a specified location
`GetNextItem`	Retrieves the next tool strip item from a specified reference point and moves it in a specified direction

Table 6.3 lists noteworthy events of the **ToolStrip** class:

Table 6.3: Noteworthy Events of the ToolStrip Class	
Event	**Description**
`AutoSizeChanged`	Occurs when the AutoSize property changes
`ItemAdded`	Occurs when a new tool strip item is added to the ToolStripItemCollection class
`ItemClicked`	Occurs when a tool strip item is clicked
`ItemRemoved`	Occurs when a tool strip item is removed from the ToolStripItemCollection class
`LayoutCompleted`	Occurs when the layout of the ToolStrip control completes
`LayoutStyleChanged`	Occurs when the LayoutStyle property changes

Note

The ToolStrip class is inherited by the MenuStrip, StatusStrip, and ContextMenuStrip classes.

The ToolStrip control contains various types of controls, such as buttons, combo boxes, labels, and text boxes, which are referred to as tool strip items. The tool strip items are instances of the **ToolStripItem** class, which exists in the **System.Windows.Forms** namespace. The individual tool strip items are instances of the classes that directly or indirectly inherit the **ToolStripItem** class. The classes that represent the tool strip items of the **ToolStrip** control are as follows:

- **ToolStripButton:** Creates a toolbar button that can contain text or an image
- **ToolStripLabel:** Creates a toolbar label that can contain text, image, and hyperlinks
- **ToolStripTextBox:** Creates a toolbar text box in which users can enter text
- **ToolStripComboBox:** Creates a toolbar combo box of the **ToolStrip** control
- **ToolStripSeparator:** Creates a toolbar vertical line that logically groups the tool strip items
- **ToolStripDropDownButton:** Creates a toolbar button that displays a drop-down list
- **ToolStripSplitButton:** Creates a combination of a standard button on the left and a drop-down button on the right

The **ToolStripItem** class and its derived classes have several properties, methods, and events that you can use to work with the tool strip items. Table 6.4 lists noteworthy properties of the **ToolStripItem** class:

Table 6.4: Noteworthy Properties of the ToolStripItem Class

Property	Description
`Alignment`	Retrieves or sets a value that signifies whether the item aligns towards the start or the end of the ToolStrip control
`Anchor`	Retrieves or sets the edges of the container to which a tool strip item is bound
`AutoSize`	Retrieves or sets a value signifying whether an item is automatically resized
`AutoToolTip`	Retrieves or sets a value indicating whether to use the Text property or the ToolTipText property for the ToolStripItem ToolTip
`Available`	Retrieves or sets a value indicating whether the tool strip item should be placed in a ToolStrip control
`CanSelect`	Retrieves a value indicating whether an item can be selected
`DisplayStyle`	Retrieves or sets a value indicating whether or not the text and images are displayed on a tool strip item
`Dock`	Retrieves or sets which tool strip item border is docked to its parent control and determines how tool strip item is resized with its parent
`Image`	Retrieves or sets the image displayed on tool strip item
`Margin`	Retrieves or sets the space between two adjacent tool strip items
`Name`	Retrieves or sets the name of a tool strip item
`Text`	Retrieves or sets the text that is to be displayed on a tool strip item
`TextDirection`	Retrieves the orientation of text used on a tool strip item
`ToolTipText`	Retrieves or sets the text that appears as a ToolTip for a control
`Visible`	Retrieves or sets a value specifying whether a tool strip item is displayed

Table 6.5 lists noteworthy methods of the **ToolStripItem** class:

Table 6.5: Noteworthy Methods of the ToolStripItem Class

Method	Description
`DoDragDrop`	Starts a drag-and-drop operation
`GetCurrentParent`	Retrieves a ToolStrip control that contains the current tool strip item
`PerformClick`	Activates a tool strip item when its Click event occurs
`Select`	Selects an item in the ToolStrip control

Table 6.6 lists noteworthy events of the **ToolStripItem** class:

Table 6.6: Noteworthy Events of the ToolStripItem Class

Event	Description
`AvailableChanged`	Occurs when there is a change in the value of the Available property
`Click`	Occurs when a user clicks a tool strip item
`DisplayStyleChanged`	Occurs when there is a change in the value of the DisplayStyle property
`OwnerChanged`	Occurs when the owner of the tool strip item changes
`TextChanged`	Occurs when the Text property of the tool strip item changes

Now that you have learned about some important properties, methods, and events of the **ToolStripItem** class, let's create a Windows Forms application named **ToolStripDemo**, where we can use some of these properties, methods, and events. Perform the following steps to create the **ToolStripDemo** application:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 integrated development environment (IDE).
2. *Select* **File→New→Project** from the menu bar to open the **New Project** dialog box.
3. *Select* **Visual C#→Windows from the Installed Templates** pane of the **New Project** dialog box.
4. *Select* **Windows Forms Application** from the right-hand side of the **Installed Templates** pane.
5. *Enter* **ToolStripDemo** as the name of the application in the **Name** text box.
6. *Enter* the path of the folder where you want to save the application in the **Location** combo box. In our case, we have selected the default location, which **is C:\Users\temp\Documents\Visual Studio 2010\Projects**.
7. *Click* the **OK** button in the **New Project** dialog box. The **ToolStripDemo** application is created.
8. *Select* the **Form1** form that appears in the designer and *press* the **F4** key to view its properties in the **Properties** window. *Set* the **Text** and **Size** properties of the **Form1** form to **ToolStripDemo** and **479, 300** respectively.
9. *Drag* a **Label** control from **Toolbox** and drop it on to the **Form1** form in the designer and set its **Text** and **Location** properties to **This is a demonstration of the ToolStrip control** and **97, 127** respectively.
10. *Drag* a **ToolStrip** control from the **Menus & Toolbars** tab of **Toolbox** and drop it on the **Form1** form in the designer. The **Form1** form now appears, as shown in Fig.C#-6.1:

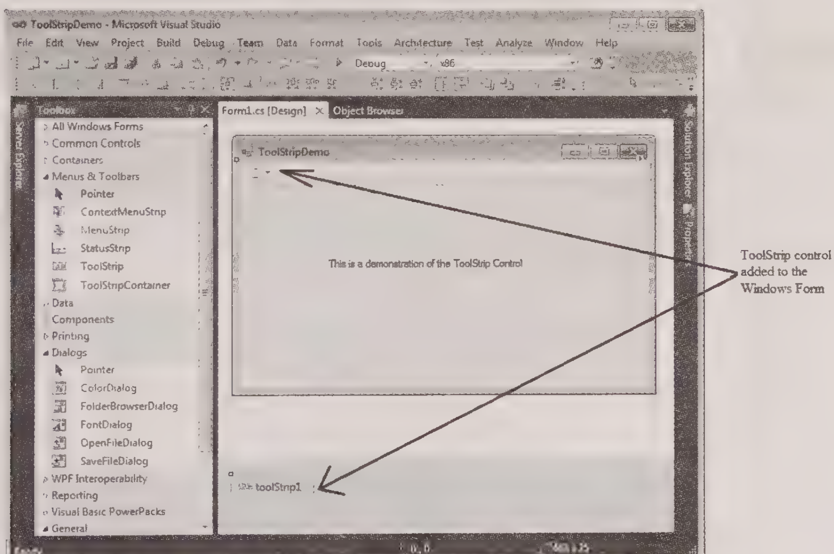


Fig.C#-6.1

Note that the **ToolStrip** control, **toolStrip1,** appears at the top of the **Form1** form as well as in the component tray, which is displayed as a rectangular strip at the bottom of the designer of the **Form1** form. This is because the **Dock** property of the **toolStrip1** control is set to **Top,** by default. You can change the **Dock** property to **Left**, **Right**, **Fill**, **Bottom**, or **None**. Now, you need to add tool strip items to the **toolStrip1** control. You can add tool strip items either at design time or at run time. Let's perform the following steps to add the tool strip items to the **toolStrip1** control at design time:

11. *Select* the **toolStrip1** control in the designer and *press* the **F4** key to open the **Properties** window.

12 *Select* the **Items** property in the **Properties** window and click the ellipsis button (...), as shown in Fig.C#-6.2:

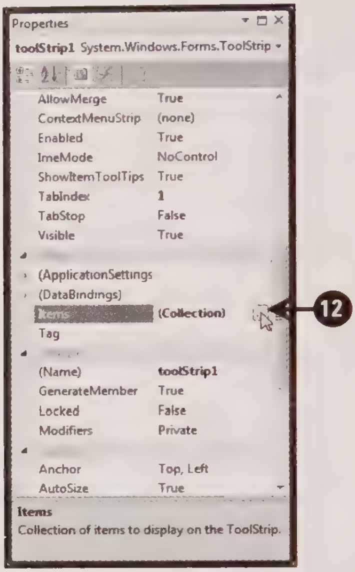


Fig.C#-6.2

The **Items Collection Editor** dialog box appears, as shown in Fig.C#-6.3:

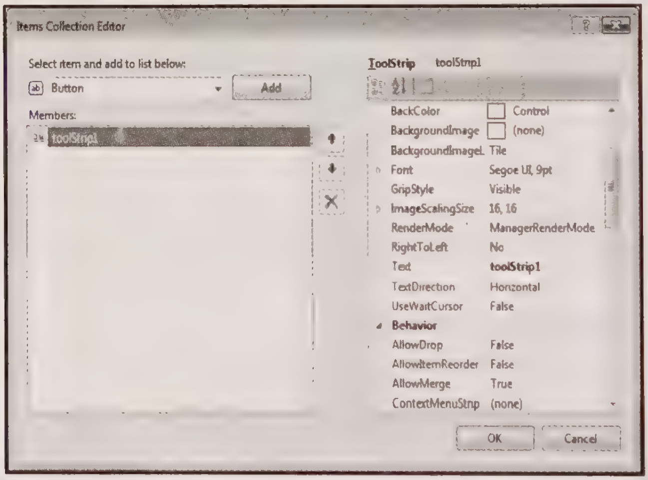


Fig.C#-6.3

In the **Items Collection Editor** dialog box, you can select the tool strip items (button, text box, label, combo box, drop-down button, or split button) to add to the **toolStrip1** control. You can also use the **Items Collection Editor** dialog box to set the properties of the tool strip items.

13 *Click* the down arrow of the drop-down list below the **Select item and add to list below** label and select the desired tool strip item, as shown in Fig.C#-6.4:

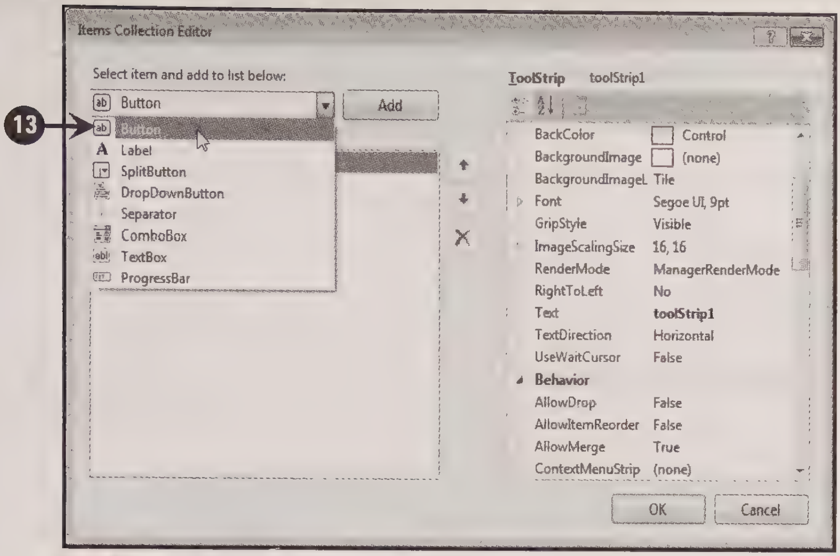


Fig.C#-6.4

In Fig.C#-6.4, you can see that we have selected a button as the tool strip item.

14 *Click* the **Add** button in the **Items Collection Editor** dialog box to add the selected tool strip item to the **tool strip1** control. The selected tool strip item is added to the list below the **Members** label, as shown in Fig.C#-6.5:

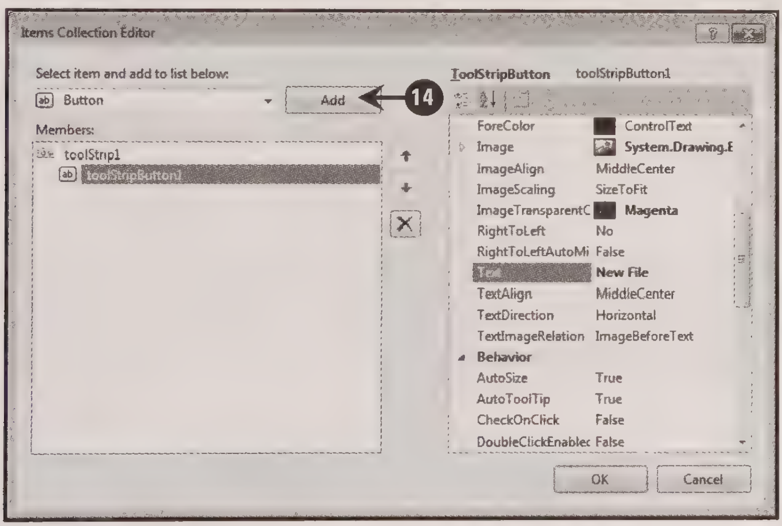


Fig.C#-6.5

After adding a tool strip item, you can set its properties in the right pane of the **Items Collection Editor** dialog box.

15 *Repeat* steps **13-14** to add more tool strip items to the **toolStrip1** control by using the **Items Collection Editor** dialog box. Table 6.7 lists the tool strip items added to the **toolStrip1** control along with their modified property values:

<table>
<tr><th colspan="4">Table 6.7: Modified Values of the Properties of the Tool Strip Items of toolStrip1</th></tr>
<tr><th>Tool Strip Item</th><th colspan="2">Property</th><th>Value</th></tr>
<tr><td><code>toolStripButton1</code></td><td colspan="2">Text</td><td>New File</td></tr>
<tr><td rowspan="2"><code>toolStripButton2</code></td><td colspan="2">Text</td><td>Open File</td></tr>
<tr><td colspan="2">DisplayStyle</td><td>ImageAndText</td></tr>
<tr><td><code>toolStripLabel1</code></td><td colspan="2">Text</td><td>ToolStrip Demo</td></tr>
<tr><td rowspan="9"><code>toolStripDropDownButton1</code></td><td colspan="2">Text</td><td>Font Size</td></tr>
<tr><td colspan="2">DisplayStyle</td><td>ImageAndText</td></tr>
<tr><td colspan="3">DropDownItems (Five menu items are added and their Text property is set to following values)</td></tr>
<tr><td>Menu Item</td><td>Property</td><td>Value</td></tr>
<tr><td>toolStripMenuItem1</td><td>Text</td><td>12</td></tr>
<tr><td>toolStripMenuItem2</td><td>Text</td><td>14</td></tr>
<tr><td>toolStripMenuItem3</td><td>Text</td><td>16</td></tr>
<tr><td>toolStripMenuItem4</td><td>Text</td><td>18</td></tr>
<tr><td>toolStripMenuItem5</td><td>Text</td><td>20</td></tr>
<tr><td><code>toolStripTextBox1</code></td><td colspan="2">Text</td><td>Enter Text</td></tr>
<tr><td rowspan="7"><code>toolStripSplitButton1</code></td><td colspan="2">Text</td><td>Edit</td></tr>
<tr><td colspan="2">DisplayStyle</td><td>ImageAndText</td></tr>
<tr><td colspan="3">DropDownItems (Three menu items are added and their Text property is set to the following values)</td></tr>
<tr><td>Menu Item</td><td>Property</td><td>Value</td></tr>
<tr><td>toolStripMenuItems6</td><td>Text</td><td>Cut</td></tr>
<tr><td>toolStripMenuItems7</td><td>Text</td><td>Copy</td></tr>
<tr><td>toolStripMenuItems8</td><td>Text</td><td>Paste</td></tr>
<tr><td rowspan="2"><code>toolStripComboBox1</code></td><td colspan="2">Text</td><td>Font</td></tr>
<tr><td colspan="2">Items</td><td>Arial
Times New Roman
Courier New
Optima
Book Antiqua
Lucida Console</td></tr>
</table>

All the tool strip items are added in the Members list (Fig.C#-6.6).

16. *Click* the **OK** button in the **Items Collection Editor** dialog box to close it, as shown in Fig.C#-6.6:

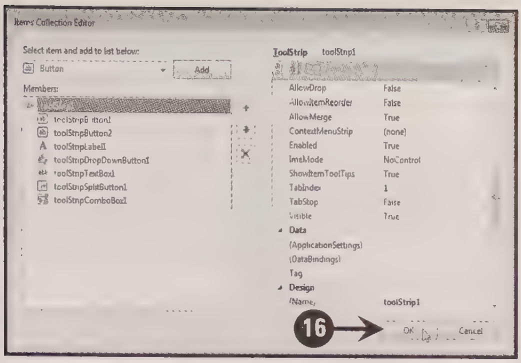


Fig.C#-6.6

The **toolStrip1** control now appears in the designer, as shown in Fig.C#-6.7:

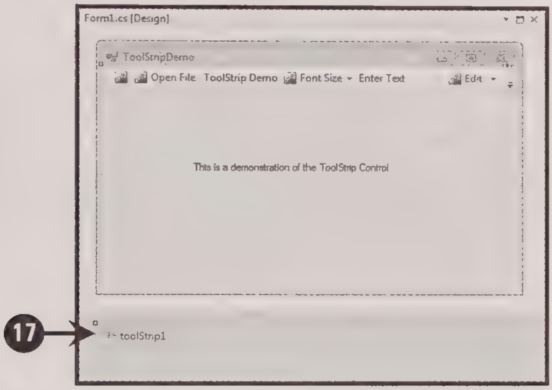


Fig.C#-6.7

17 *Double-click* the **toolStrip1** control and *add* the highlighted code, given in Listing 6.1, in its **ItemClicked** event in the code-behind file (**Form1.cs** file):

Listing 6.1: Showing the Code to Add in the toolStrip1_ItemClicked Event

```
private void toolStrip1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)
{
        MessageBox.Show("You clicked " + e.ClickedItem.Name);
}
```

In Listing 6.1, the **e.ClickedItem.Name** property retrieves the name of the tool strip item that a user clicks.

18 *Press* the **F5** key to run the **ToolStripDemo** application. The output appears, as shown in Fig.C#-6.8:

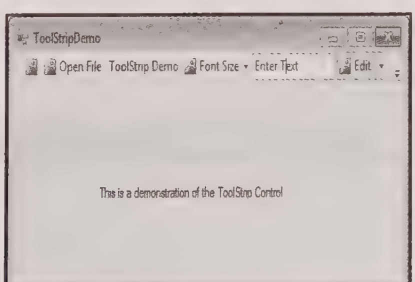


Fig.C#-6.8

19 *Click* any button, drop-down button, split button, or label on the toolbar, a message box appears displaying the name of the tool strip item clicked, as shown in Fig.C#-6.9:

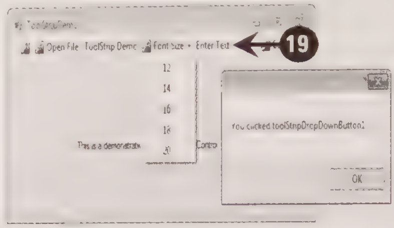


Fig.C#-6.9

In Fig.C#-6.9, when the **Font Size** item is clicked on the toolbar, a message box appears, stating **You clicked toolStripDropDownButton1**.

Using the MenuStrip Control

While working with the Visual Studio 2010 IDE, you must have several times used the menu bar and its various menus. You can also design a menu bar for your own Windows Forms application to logically group certain actions and options that users can perform and select, respectively. You can add a menu bar to your Windows Forms application by using the **MenuStrip** control.

The **MenuStrip** control acts as a container for a single menu bar on a Windows form. The menu bar contains several menus that have logically grouped menu items. The menu items grouped under a menu bar represent options that a user can select. The menu items can further have submenus, which is another menu with a different set of options.

The **MenuStrip** control is an instance of the **MenuStrip** class, which exists in the **System.Windows.Forms** namespace.

Table 6.8 lists noteworthy properties of the **MenuStrip** class:

Table 6.8: Noteworthy Properties of the MenuStrip Class

Property	Description
`CanOverflow`	Retrieves or sets a value that indicates whether the MenuStrip control supports overflow functionality.
`GripStyle`	Retrieves or sets the visibility of the grip of the MenuStrip control. The GripStyle property is used to reposition the MenuStrip control. For instance, if the GripStyle property of a MenuStrip control is set to Hidden, then a user cannot reposition the MenuStrip control.
`ShowItemToolTips`	Retrieves or sets a value that indicates whether ToolTips appear for the MenuStrip control
`Stretch`	Retrieves or sets a value that indicates whether the MenuStrip control stretches from end to end in its container

Table 6.9 lists noteworthy events of the **MenuStrip** class:

Table 6.9: Noteworthy Events of the MenuStrip Class

Events	Description
`MenuActivate`	Occurs when a user accesses or opens a menu from the menu bar using either a mouse or keyboard
`MenuDeactivate`	Occurs when the MenuStrip control is deactivated

Note that the **MenuStrip control** is only a container for the menus. The menus and menu items are mostly represented by the **ToolStripMenuItem** objects, which are instances of the **ToolStripMenuItem** class. This class

allows you to perform various operations with the menu items and submenus, such as adding menu items to menus and submenus, adding menu separators (a horizontal line that separates two or more menus), specifying shortcut keys for menus and menu items, and displaying check marks.

Table 6.10 lists noteworthy properties of the **ToolStripMenuItem** class:

Table 6.10: Noteworthy Properties of the ToolStripMenuItem Class

Property	Description
`Checked`	Retrieves or sets a value that indicates whether the ToolStripMenuItem object is checked
`CheckOnClick`	Retrieves or sets a value specifying whether the ToolStripMenuItem object automatically appears checked or unchecked when clicked
`CheckState`	Retrieves or sets a value that indicates whether the ToolStripMenuItem object is in the checked, unchecked, or indeterminate state
`Enabled`	Retrieves or sets a value that indicates whether a child menu item in a menu is enabled
`ShortcutKeys`	Retrieves or sets the shortcut keys associated with the ToolStripMenuItem object
`ShowShortcutKeys`	Retrieves or sets a value that indicates whether the shortcut keys associated with the ToolStripMenuItem object are displayed next to it

Table 6.11 lists noteworthy events of the **ToolStripMenuItem** class:

Table 6.11: Noteworthy Events of the ToolStripMenuItem Class

Event	Description
`CheckedChanged`	Occurs when the Checked property changes
`CheckStateChanged`	Occurs when the CheckState property changes

To create submenus, you can add multiple **ToolStripMenuItem** objects to an existing **ToolStripMenuItem** object.

Tip

There are various menu conventions in Windows that you should adhere to if you are going to release your programs for public use. These menu conventions are as follows:

- *If a menu item opens a dialog box, you should add an ellipsis (...) after its name (such as Print...).*
- *There are many standard shortcuts already in use, such as CTRL+S for Save, CTRL+X for Cut, CTRL+V for Paste, and CTRL+C for Copy. Therefore, do not assign any different shortcuts for these functions.*
- *There are certain menus in the menu bar that do not open a menu but instead perform some action immediately. Therefore, similar immediate actions should have an exclamation mark (!) after their names (such as Connect!).*
- *The File menu should always be the first menu and an Exit item, the last item of the menu.*

Now, let's create a Windows Forms application named **MenuStripDemo**, where we use some of the properties and events of the **ToolStripMenuItem** class. Perform the following steps to create the **MenuStripDemo** application:

1. *Repeat* steps **1-4** under the **Using the ToolStrip Control** heading of this chapter.
2. *Enter* **MenuStripDemo** as the name of the application in the **Name** text box of the **New Project** dialog box and *enter* the path of the folder where you want to save the application in the **Location** combo box,. In our case, we have selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects.**
3. *Click* the **OK** button in the **New Project** dialog box. The **MenuStripDemo** application is created.

4. *Select* the **Form1** form in the designer and set its **Text** property to **MenuStripDemo** in the **Properties** window.
5. *Drag* a **Button** control from **Toolbox** and drop it on to the **Form1** form in the designer and set its **Text**, **Location**, and **Size** properties to **Disable** a **Menu Item**, **67**, **182**, and **141**, **23** respectively.
6. *Drag* a **MenuStrip** control from the **Menus & Toolbars** tab of **Toolbox** and drop it on the **Form1** form in the designer.

   A **MenuStrip** control, **menuStrip1**, is added at the top of **Form1** form as well as in the component tray.
7. *Select* the **menuStrip1** control. You see that a text box with **Type Here** is displayed, as shown in Fig.C#-6.10:

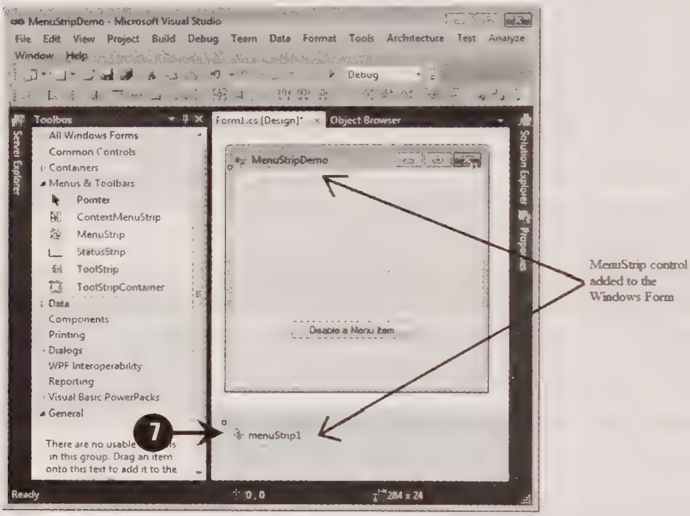


Fig.C#-6.10

8. *Enter* the **name** of the first menu in the text box. In this case, we have entered **File** (Fig.C#-6.11).
9. *Click* the text box where you have entered the text, **File**. You see that two other text boxes appear—one below the first text box and the other on the right of the first text box, as shown in Fig.C#-6.11:

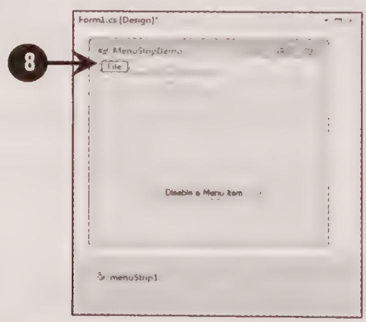


Fig.C#-6.11

The text box below the first one allows you to add a menu item to the first menu.

10. *Enter* **New** in the text box that appears below the text box with the text **File**, as shown in Fig.C#-6.12:

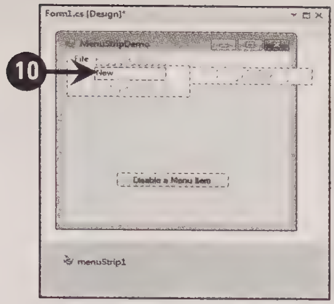


Fig.C#-6.12

11 *Enter* **Open** and **Exit** in the text boxes that appear below the text box with the text, **New**.

Note that the text that you type in the text boxes refers to the value of the **Text** property of the menu or menu item.

12 *Click* the **Type Here** text box next to the **File** menu to create the second menu, as shown in Fig.C#-6.13:

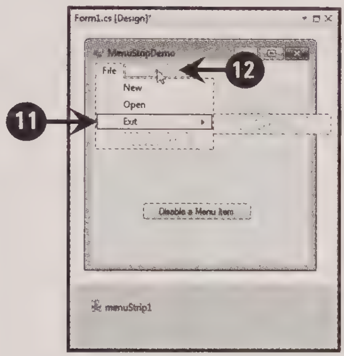


Fig.C#-6.13

13 *Enter* the name of the second menu. In our case, we have named the second menu as **Format**.

14 *Add* menu items to the **Format** menu by typing their names in the **Type Here** text boxes that appear below the **Format** menu. In our case, we have named **Font** as the first menu item of the **Format** menu.

15 *Click* and *enter* the name of submenu items in the text box that appears to the right of the existing menu items, to create a submenu for an existing menu item. In our case, we have created a submenu for the **Font** menu item. The submenu items include **Arial**, **Times New Roman**, and **Courier New**.

Similarly, we have added another menu item named **Font Weight** to the **Format** menu. The **Font Weight** menu item has a submenu with **12** and **14** as its items, as shown in Fig.C#-6.14:

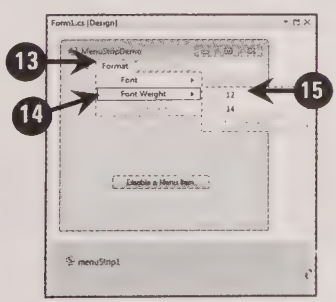


Fig.C#-6.14

16 *Select* a menu item in the designer, to set shortcut keys, and *press* the **F4** key to open the **Properties** window. In our case, we have selected the **New** menu item of the **File** menu (Fig.C#-6.15).

You can set shortcut keys for a menu item with the help of an access key in combination to CTRL, ALT, or SHIFT keys.

Tip

An access key is a character that you can use with the ALT key to access or select either a menu or menu item. To specify an access key for a menu item, you need to prefix the desired character with an ampersand (&) in the Text property of the menu item. The access key appears underlined in the menu. For example, if you specify the Text property of a menu item as E&xit, then x becomes the access key for the Exit menu item.

17 *Select* the **ShortcutKeys** property and click the down arrow in the **Properties** window (Fig.C#-6.15).

18 *Select* the **Ctrl** check box (Fig.C#-6.15).

19 *Click* the down arrow of the drop-down list below the **Key label**, as shown in Fig.C#-6.15:

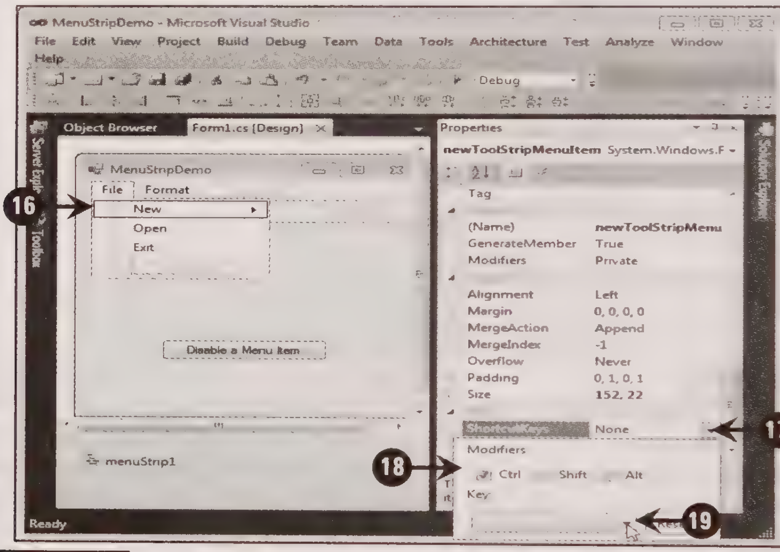


Fig.C#-6.15

A drop-down list with all the available characters appears (Fig.C#-6.16).

20 *Select* the desired character from the drop-down list. In our case, we have selected letter **N**, as shown in Fig.C#-6.16:

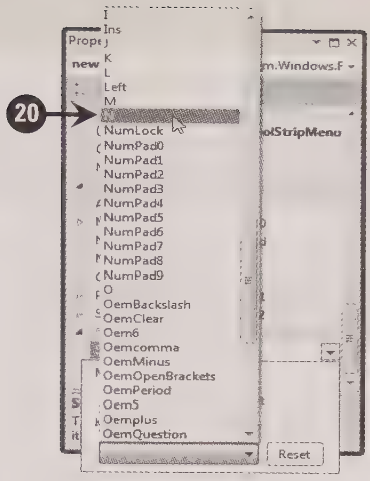


Fig.C#-6.16

21 *Press* the **ENTER** key. The shortcut keys appear next to the corresponding menu item, as shown in Fig.C#-6.17:

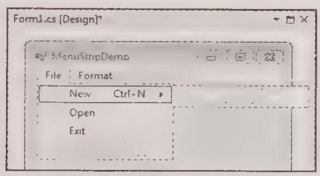


Fig.C#-6.17

22 *Double-click* the **Form1** form and *add* the highlighted code, given in Listing 6.2, in its **Load** event in the code-behind file (**Form1.cs** file):

Listing 6.2: Showing the Code to Add in the Form1_Load Event

```
private void Form1_Load(object sender, EventArgs e)
{
        arialToolStripMenuItem.CheckOnClick = true;
        timesNewRomanToolStripMenuItem.CheckOnClick = true;
        courierNewToolStripMenuItem.CheckOnClick = true;
        toolStripMenuItem2.CheckOnClick = true;
        toolStripMenuItem3.CheckOnClick = true;
}
```

In Listing 6.2, the **CheckOnClick** property of the **arialToolStripMenuItem**, **timesNewRomanToolStripMenuItem**, **courierNewToolStripMenuItem**, **toolStripMenuItem2**, and **toolStripMenuItem3** menu items are set to **true**. This implies that if you click any of these menu items at run time, a check mark appears on the left of the menu item.

23 *Switch* to the designer and *double-click* the **menuStrip1** control and *add* the highlighted code, given in Listing 6.3, in the **ItemClicked** event in the code-behind file:

Listing 6.3: Showing the code to Add in the ItemClicked Event

```
private void menuStrip1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)
{
        MessageBox.Show("You clicked " + e.ClickedItem.Text + " menu");
}
```

24 *Switch* to the designer and *double-click* the **button1** button.

25 *Add* the highlighted code given in Listing 6.4 in the **Click** event of **button1**:

Listing 6.4: Showing the Code to Add in the button1_Click Event

```
private void button1_Click(object sender, EventArgs e)
{
        fontToolStripMenuItem.Enabled = false;
      MessageBox.Show("Font menu item disabled");
}
```

26 *Press* the **F5** key to run the **MenuStripDemo** application. The output appears, as shown in Fig.C#-6.18:

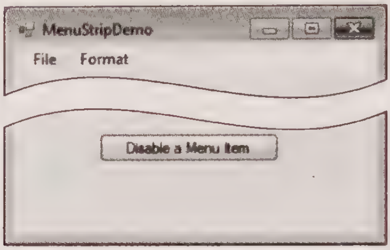


Fig.C#-6.18

27 *Click* a menu on the menu strip to open it, a message box appears. In our case, we have clicked the **Format** menu, which displays a message box with the **message, You clicked Format menu** (Fig.C#-6.19).

28 *Click* the **OK** button in the message box to close it, as shown in Fig.C#-6.19:

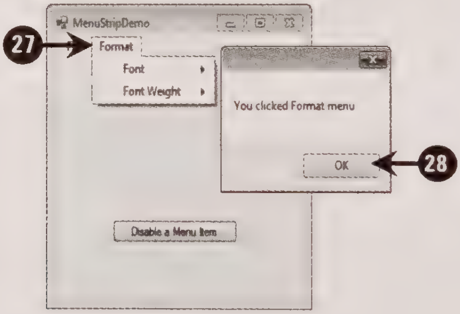


Fig.C#-6.19

29 *Select* one of the menu items from the **Font** sub menu. In our case, we have selected **Times New Roman**.

Once a menu item is selected, a check mark appears on its left when you open the **Font** submenu next time, as shown in Fig.C#-6.20:

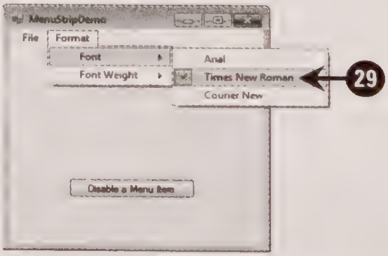


Fig.C#-6.20

Let's now learn about the **StatusStrip** control.

Using the StatusStrip Control

The **StatusStrip** control is an improved version of the **StatusBar** control. It allows you to display information about other controls in a form. For instance, you can use the **StatusStrip** control to display the name and description of a selected control, the progress of an ongoing operation, or instructions for the users.

The **StatusStrip** control is an instance of the **StatusStrip** class that exists in the **System.Windows.Forms** namespace.

Table 6.12 lists noteworthy properties of the **StatusStrip** class:

Table 6.12: Noteworthy Properties of the StatusStrip Class

Property	Description
`Dock`	Retrieves or sets the border of the StatusStrip control that is docked to its parent control
`GripStyle`	Retrieves or sets the visibility of the grip of a StatusStrip control. The GripStyle property is used to reposition the StatusStrip control. For instance, if the GripStyle property of a StatusStrip control is set to Hidden, then there is no way for a user to drag the StatusStrip control from the position it has been originally placed.
`LayoutStyle`	Retrieves or sets a value that indicates the manner in which the status strip items are placed in the StatusStrip control
`ShowItemToolTips`	Retrieves or sets a value that indicates whether the ToolTips are displayed for the StatusStrip control
`SizingGrip`	Retrieves or sets a value that indicates whether a sizing handle, such as a grip appears in the lower-right corner of the control
`Stretch`	Retrieves or sets a value that indicates whether the StatusStrip control stretches from one end to another of its container

As the **StatusStrip** class inherits the **ToolStrip** class, the **StatusStrip** control can contain several controls or items. The controls that a **StatusStrip** control can have are as follows:

- **ToolStripStatusLabel:** Refers to a label in the **StatusStrip** control
- **ToolStripDropDownButtton**: Refers to a drop-down button which when clicked displays a drop-down list
- **ToolStripSplitButton:** Refers to a split button (combination of a standard button on the left and a drop-down button on the right)
- **ToolStripProgressBar:** Refers to a progress bar

Note

By default, the StatusStrip control does not contain panels. You can add panels to a StatusStrip control by using the Items.AddRange() method.

Now, let's create a Windows Forms application named **StatusStripDemo** to see how a **StatusStrip** control works. Perform the following steps to create the **StatusStripDemo** application:

1. *Repeat* steps **1-4** under the **Using the ToolStrip Control** heading of this chapter.
2. *Enter* **StatusStripDemo** as the name of the application in the **Name** text box of the **New Project** dialog box and *enter* the path of the folder where you want to save the application in the **Location** combo box.
3. *Click* the **OK** button in the **New Project** dialog box. The **StatusStripDemo** application is created.
4. *Select* the **Form1** form in the designer and *press* the **F4** key to open its **Properties** window. In the **Properties** window, set the **Text** and **Size** properties of the **Form1** form to **StatusStripDemo** and **450,300** respectively.
5. *Drag* **Label** and **TextBox** controls from **Toolbox** and drop them on the **Form1** form. Set the properties of the Label **(label1)** and TextBox **(textBox1)** controls, as given in Table 6.13:

Table 6.13: Values of the Properties of Label and Text Box in Form1		
Control	**Property**	**Value**
`label1`	Text	Demonstration of the StatusStrip Control
	Location	126, 23
`textBox1`	Multiline	True
	Location	129, 54

6 *Drag* a **StatusStrip** control from the **Menus & Toolbars** tab of **Toolbox** and *drop* it on to the **Form1** form. The **StatusStrip** control, **stausStrip1**, is added in the**Form1** form, as shown in Fig.C#-6.21:

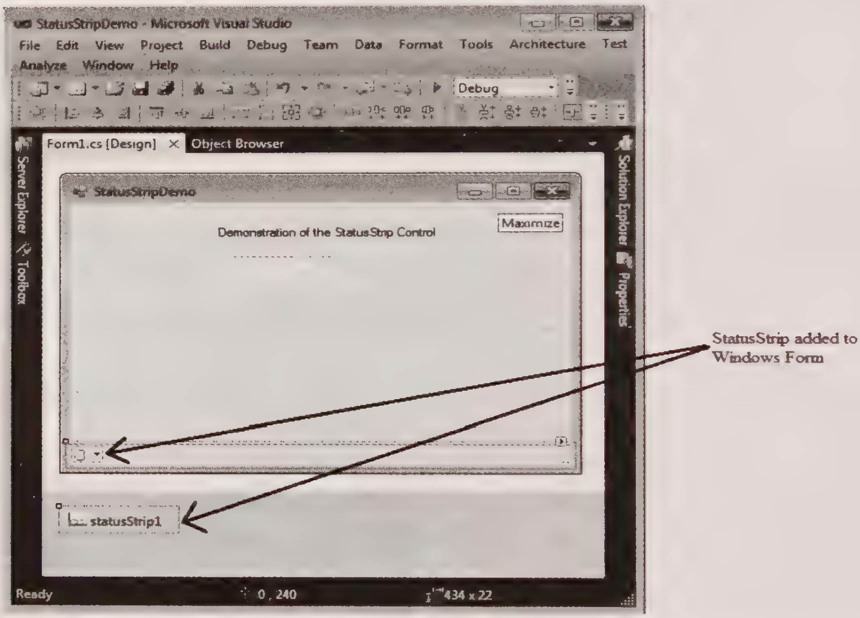


Fig.C#-6.21

7 *Select* the **statusStrip1** control in the designer and *press* the **F4** key to open the **Properties** window.

8 *Select* the **Items** property and *click* the ellipsis button (**...**), as shown in Fig.C#-6.22:

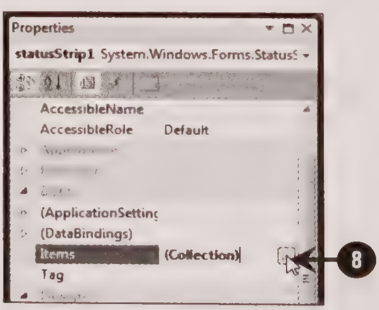


Fig.C#-6.22

The **Items Collection Editor** dialog box appears (Fig.C#-6.23).

9 *Select* the type of control that you want to add to the **statusStrip1** control from the drop-down list below the **Select item and add to list below** label. In our case, we have selected the **StatusLabel** control (Fig.C#-6.23).

10 *Click* the **Add** button to add the selected item to the **Members** list, as shown in Fig.C#-6.23:

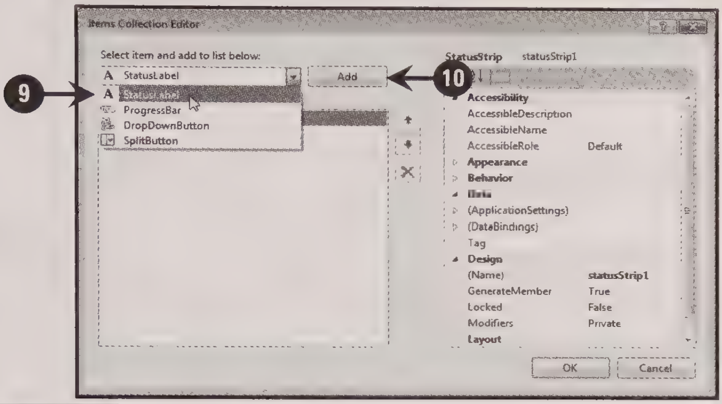


Fig.C#-6.23

The **StatusLabel** control is added to the **Members** list, as shown in Fig.C#-6.24:

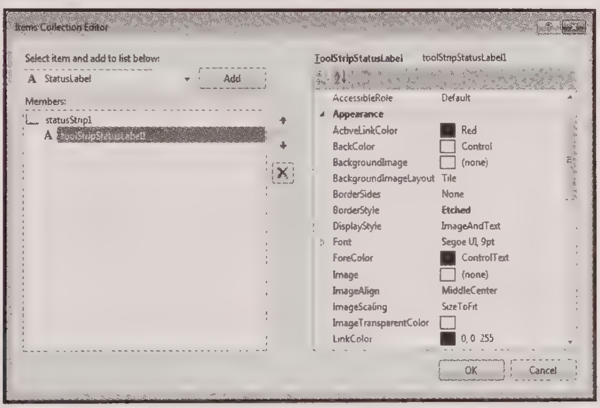


Fig.C#-6.24

11 *Similarly, add* a drop-down button to the **statusStrip1** control (Fig.C#-6.25).

12 *Select* the **toolStripDropDownButton1 control** from the **Members** pane (Fig.C#-6.25).

13 *Click* the **DisplayStyle** drop-down list in the **Properties** window and select the **ImageAndText** option from the drop-down list to display both image and text in the **toolStripDropDownButton1** control (Fig.C#-6.25).

14 *Set* the **Text** property of the **toolStripDropDownButton1** control as **Alignment** (Fig.C#-6.25).

15 *Click* the **OK** button in the **Items Collection Editor** dialog box, as shown in Fig.C#-6.25:

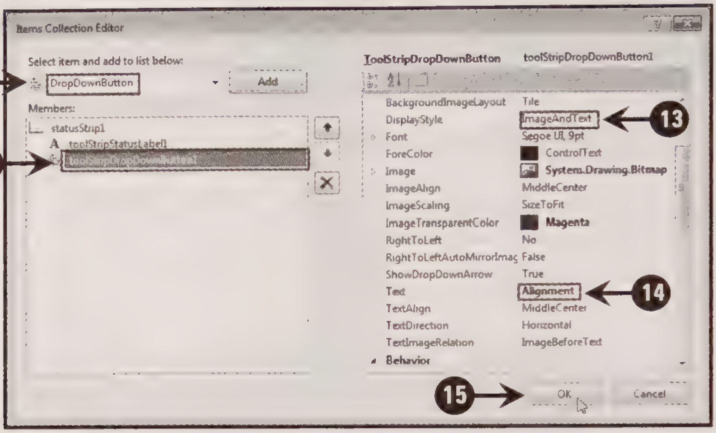


Fig.C#-6.25

The selected controls are added to the **statusStrip1** control (Fig.C#-6.26).

16 *Double-click* the **textBox1** control, as shown in Fig.C#-6.26:

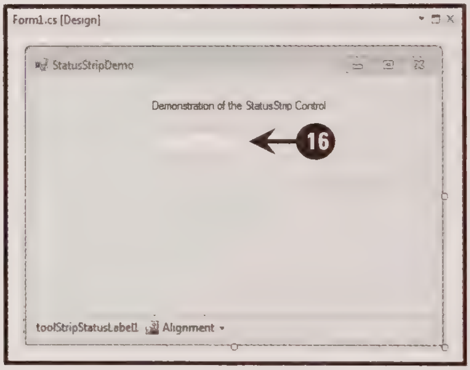


Fig.C#-6.26

17 *Add* the highlighted code, given in Listing 6.5, in the **TextChanged** event of textBox1:

Listing 6.5: Showing the Code of textBox1_TextChanged Event

```
private void textBox1_TextChanged(object sender, EventArgs e)
{
        string[] str = textBox1.Text.Split(' ','\n');
      toolStripStatusLabel1.Text  =  ("Words:  " + str.Length + ",  " +
        "Characters                                                    (with
        spaces): " + textBox1.TextLength);
}
```

18 *Press* the **F7** key to switch to the designer and *double-click* the **statusStrip1** control. *Add* the highlighted code, given in Listing 6.6, in the **ItemClicked** event of the **statusStrip1** control:

Listing 6.6: Showing the Code of statusStrip1_ItemClicked Event

```
private void statusStrip1_ItemClicked(object sender, ToolStripItemClickedEventArgs e)
{
        MessageBox.Show("Item clicked: " + e.ClickedItem.Name);
}
```

19 *Press* the **F5** key to run the **StatusStripDemo** application and *enter* some text in the text box, as shown in Fig.C#-6.27:

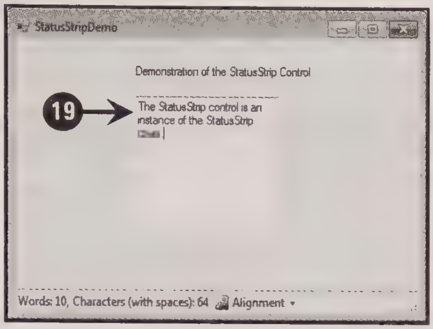


Fig.C#-6.27

In Fig.C#-6.27, you can see that the number of words and characters in the text is displayed in the first item, **toolStripStatusLabel1** of the status strip, **statusStrip1**.

20 *Click* either of the status strip items (the label or the drop-down button), a message box displaying the name of the tool strip item appears, as shown in Fig.C#-6.28:

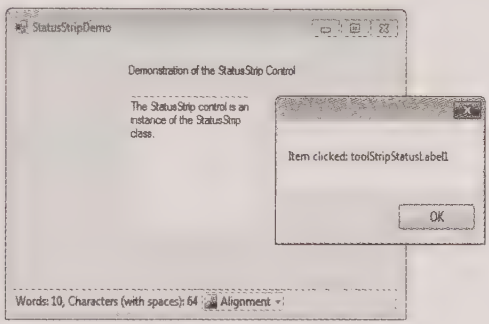


Fig.C#-6.28

Now, let's learn how to work with dialog boxes in C# 2010.

Working with Dialog Boxes

C# 2010 provides you various controls, such as **FolderBrowserDialog and OpenFileDialog,** to work with built-in dialog boxes, such as **Browse for Folder** and **Open** dialog boxes. The dialog boxes provide you with a UI that facilitates interaction between the users and the applications. They contain different controls that represent various options the user can select and use to perform a particular task. Basically, there are the following two types of dialog boxes:

- **Modal dialog box:** Retrieves additional information from a user. The user cannot proceed further or open the main window in an application till the modal dialog box is open. Generally, the **OK** button in the modal dialog box is used to indicate that the user has finished entering the data while the **Cancel** button is used to terminate the function. **Open**, **Print**, and **Save As** dialog boxes are some of the most common examples of modal dialog boxes.
- **Modeless dialog box:** Does not prevent a user from activating other windows in an application as immediate input is not required from the user to proceed further. **The Find and Replace** dialog box in Microsoft Word is an example of a modeless dialog box.

All the dialog boxes available for Windows Forms applications inherit the **System.Windows.Forms.CommonDialog** class. This class provides the **ShowDialog()** method that displays a

dialog box and returns the result of the dialog box. The result of a dialog box refers to a value from the **System.Windows.Forms.DialogResult** enumeration. The value of the **DialogResult** enumeration corresponds to the button that the user clicks in the dialog box. Table 6.14 lists the values of the **DialogResult** enumeration:

Table 6.14: Values of the DialogResult Enumeration

Enumeration Value	Description
`OK`	Corresponds to the OK button in a dialog box
`Cancel`	Corresponds to the Cancel button in a dialog box
`Yes`	Corresponds to the Yes button in a dialog box
`No`	Corresponds to the No button in a dialog box
`Abort`	Corresponds to the Abort button in a dialog box
`Retry`	Corresponds to the Retry button in a dialog box
`Ignore`	Corresponds to the Ignore button in a dialog box
`None`	Returns nothing and implies that the dialog box remains visible unless closed explicitly

Although every built-in dialog box available in C# 2010 for Windows Forms applications performs a different task, all dialog boxes support the **ShowDialog()** method

and the **DialogResult** enumeration. This implies that you can use the **ShowDialog()** method to display any of the built-in dialog boxes and the **DialogResult** enumeration to hold the result of the dialog boxes.

In Windows Forms applications, the built-in dialog boxes are provided as controls. The common built-in dialog box controls are available in the **Dialogs** tab of **Toolbox**. The dialog box controls that pertain to printing are available in the **Printing** tab of **Toolbox**. Some of the commonly used dialog box controls are as follows:

- **FolderBrowserDialog** control
- **OpenFileDialog** control
- **SaveFileDialog** control
- **PrintDocument** control
- **PrintDialog** control

Let's discuss these in detail.

Using the FolderBrowserDialog Control

The **FolderBrowserDialog** control allows you to work with the **Browse For Folder** dialog box that provides you the facility to browse through folders (not files), create new folders, and select folders to view the files. It displays the available physical drives and the folders in those drives in a hierarchical tree structure.

The **FolderBrowserDialog** control is an instance of the **FolderBrowserDialog** class. Table 6.15 lists noteworthy properties of the **FolderBrowserDialog** class:

Table 6.15: Noteworthy Properties of the FolderBrowserDialog Class

Property	Description
`Description`	Retrieves or sets the descriptive text displayed above the hierarchical tree in the Browse For Folder dialog box.
`RootFolder`	Retrieves or sets the root folder from where the browsing starts. By default, the desktop is selected as the root location for browsing.
`SelectedPath`	Retrieves or sets the path selected by a user.
`ShowNewFolderButton`	Retrieves or sets a value indicating whether the **Browse For Folder** dialog box allows the users to create new folders.

Let's create a Windows Forms application named **FolderBrowserDialogDemo** where we use some of the properties of the **FolderBrowserDialog** class. Perform the following steps to create the **FolderBrowserDialogDemo** application:

1. *Repeat* steps **1-4** under the **Using the ToolStrip Control** heading of this chapter.
2. *Enter* **FolderBrowserDialogDemo** as the name of the application in the **Name** text box and *enter* the path of the folder where the application will be saved in the **Location** combo box.
3. *Click* the **OK** button in the **New Project** dialog box. The **FolderBrowserDialogDemo** application is created.
4. *Select* the **Form1** form and *press* the **F4** key to open the **Properties** window. In the **Properties** window, *set* the **Text** property of the **Form1** form to **FolderBrowserDialogDemo**.
5. *Drag* two **Label**, a **Button**, and a **ListBox** controls from **Toolbox** and drop them on the **Form1** form. *Set* the properties of the Label controls (**label1** and **label2**), Button (**button1**), and ListBox (**listBox1**) controls, as given in Table 6.16:

Table 6.16: Values of the Properties of the Labels, Buttons, and Text Box in Form1

Control	Property	Value
label1	Text	Click the button to select a folder
	Location	12, 20
label2	Text	Recently selected folders are:
	Location	12, 119
button1	Text	Browse
	Location	87, 75
	Size	104, 23
listBox1	Location	15, 145
	Size	257, 95

6. *Drag* the **FolderBrowserDialog** control from the **Dialogs** tab of **Toolbox** and *drop* it on **Form1**, as shown in Fig.C#-6.29:

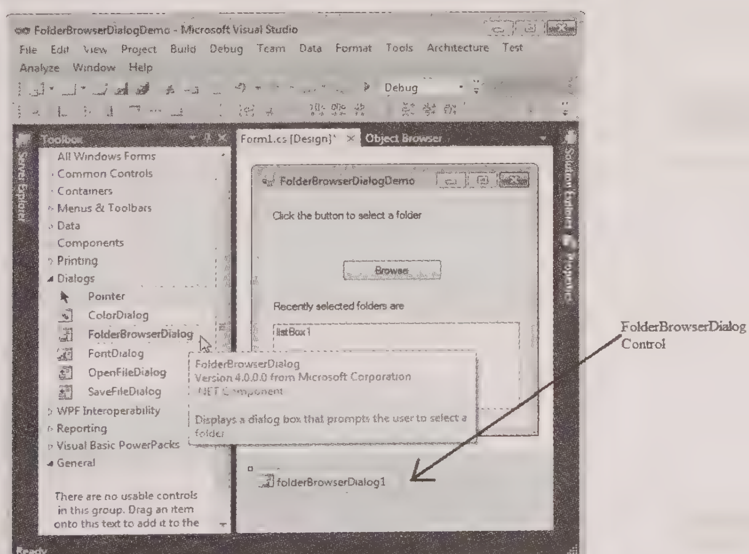


Fig.C#-6.29

In Fig.C#-6.29, you can see that when you drop the **FolderBrowserDialog** control on the designer, it appears in the component tray.

7. *Double-click* the **Browse** button in the designer and *add* the highlighted code, given in Listing 6.7, in its **Click** event in the code-behind file (**Form1.cs** file):

Listing 6.7: Showing the Code of the button1_Click Event

```
private void button1_Click(object sender, EventArgs e)
{
   folderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer;
 if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
 {
   if (listBox1.Items.Count >=5)
               listBox1.Items.RemoveAt(0);
       listBox1.Items.Add(folderBrowserDialog1.SelectedPath);
 }
}
```

In Listing 6.7, the **RootFolder** property of the **folderBrowserDialog1** control is set to **Environment.SpecialFolder.MyComputer**. This implies that users can start browsing from the **Computer** folder. You observe that when you select a folder and click the **OK** button in the dialog box, the path of the selected folder is added and displayed in the list box.

8. *Press* the **F5** key to run the **FolderBrowserDialogDemo** application. The output appears (Fig.C#-6.30).

9. *Click* the **Browse** button, as shown in Fig.C#-6.30:

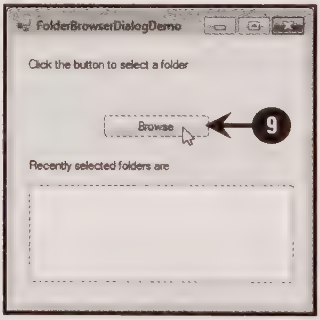


Fig.C#-6.30

The **Browse For Folder** dialog box appears, as shown in Fig.C#-6.31:

Fig.C#-6.31

In Fig.C#-6.31, you can see that the root folder for browsing is the Computer folder.

10 *Browse* and *select* any of the folders (Fig.C#-6.32).

11 *Click* the **OK** button in the **Browse For Folder** dialog box, as shown in Fig.C#-6.32:

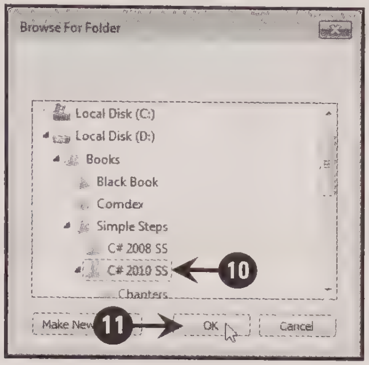


Fig.C#-6.32

The path of the selected folder is added to the list box, as shown in Fig.C#-6.33:

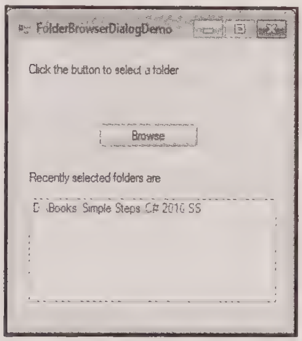


Fig.C#-6.33

12 Similarly, *browse* and *select* other folders by clicking the **Browse** button to open the **Browse For Folder** dialog box as many times as required, as shown in Fig.C#-6.34:

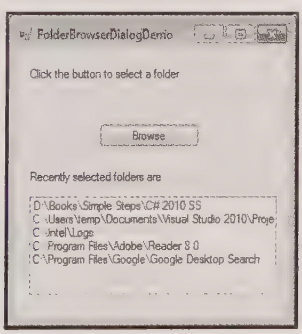


Fig.C#-6.34

In Fig.C#-6.34, the list box displays the path of five selected folders. You can also create a new folder by clicking the **Make New Folder** button in the **Browse For Folder** dialog box, as shown in Fig.C#-6.35:

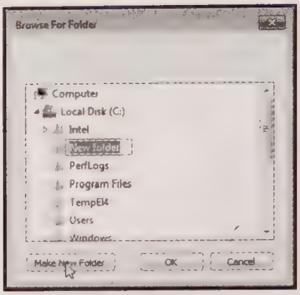


Fig.C#-6.35

Next, let's discuss about the **OpenFileDialog** control.

Using the OpenFileDialog Control

The **OpenFileDialog** control displays the **Open** dialog box, which prompts a user to select an existing file to view it. It allows the user to specify the path of an existing file that the user wants to open. The **OpenFileDialog** control encapsulates the functionality to first check whether the file that a user wants to open exists on the computer; if it exists, the control allows the user to open it.

The **OpenFileDialog** control is an instance of the **OpenFileDialog** class, which inherits the **FileDialog** class.

Table 6.17 lists noteworthy properties of the **OpenFileDialog** class:

Table 6.17: Noteworthy Properties of the OpenFileDialog Class

Property	Description
`CheckFileExists`	Retrieves or sets a value indicating whether the Open dialog box displays a warning in case a user specifies the name of a non-existent file
`ShowReadOnly`	Retrieves or sets a value signifying whether the Open dialog box displays a read-only check box
`ReadOnlyChecked`	Retrieves or sets a value signifying whether the read-only check box is selected in the Open dialog box
`Multiselect`	Retrieves or sets a value signifying whether the Open dialog box allows multiple file selection
`SafeFileName`	Retrieves the file name (not the path) and extension of the selected file in the Open dialog box
`SafeFileNames`	Retrieves an array of file names (not the path) and extensions for all the selected files in the Open dialog box

Table 6.18 lists noteworthy methods of the **OpenFileDialog** class:

Table 6.18: Noteworthy Methods of the OpenFileDialog Class

Method	Description
`OpenFile`	Opens the selected file with read-only permissions
`Reset`	Resets all options to their default values

Let's create a simple Windows Forms application named **OpenFileDialogDemo** to use the **OpenFileDialog** control. Perform the following steps to create **OpenFileDialogDemo**:

1. *Repeat* steps **1-4** under the **Using the ToolStrip Control** heading of this chapter.

2. *Enter* **OpenFileDialogDemo** as the name of the application in the **Name** text box of the **New Project** dialog box. *Enter* the path of the folder where you want to save the application in the **Location** combo box.
3. *Click* the **OK** button in the **New Project** dialog box. The **OpenFileDialogDemo** application is created.

   Select the **Form1** form in the designer and *press* the **F4** key to open the **Properties** window. In the **Properties** window, *set* the **Text** property of the **Form1** form to **OpenFileDialogDemo**.
5. *Drag* **Label**, **Button**, and **PictureBox** controls from **Toolbox** and *drop* them on the **Form1** form. *Set* the properties of the Label (**label1**), Button (**button1**), and PictureBox (**pictureBox1**) controls to the values given in Table 6.19:

Table 6.19: Values of the Properties of the Label, Button, and Picture Box in Form1

Control	Property	Value
label1	Text	Image file:
	Location	12, 19
button1	Text	Open Image
	Location	29, 216
	Size	227, 29
pictureBox1	Location	15, 59
	Size	257, 130

6. *Drag* the **OpenFileDialog** control from the **Dialogs** tab of **Toolbox** and drop it on the **Form1** form. The **OpenFileDialog** control appears in the component tray, as shown in Fig.C#-6.36:

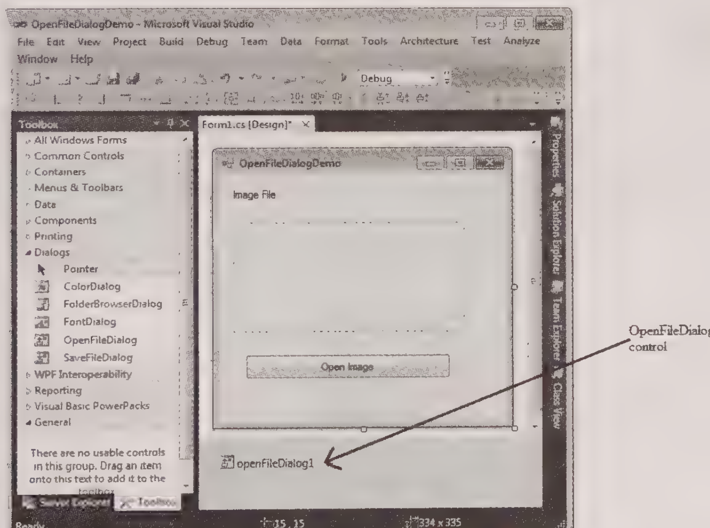


Fig.C#-6.36

7. *Double-click* the **Open Image** button and *add* the highlighted code, given in Listing 6.8, in its **Click** event:

Listing 6.8: Showing the Code of the button1_Click Event

```
private void button1_Click(object sender, EventArgs e)
{
        openFileDialog1.Multiselect = false;
      if (openFileDialog1.ShowDialog() != DialogResult.Cancel)
      {
```

```
            pictureBox1.SizeMode = PictureBoxSizeMode.Zoom;
            pictureBox1.Image = Image.FromFile(openFileDialog1.FileName);
        }
        label1.Text += " " +openFileDialog1.SafeFileName;
}
```

In Listing 6.8, the **Multiselect** property of the **openFileDialog1** control is set to **false**, which implies that a user can select only one file at a time. The **if** statement checks whether the result of the **Open** dialog box is not **Cancel**, that is, the user does not click the **Cancel** button in the dialog box. If the user does not click the **Cancel** button, the picture box displays the image file that the user has selected.

8 *Press* the **F5** key to run the application. The output appears (Fig.C#-6.37).

9 *Click* the **Open Image** button, as shown in Fig.C#-6.37:

Fig.C#-6.37

The **Open** dialog box appears.

10 *Select* an image file in the **Open** dialog box. In our case, we have selected a file named **Koala**.

11 *Click* the **Open** button in the **Open** dialog box.

The selected image file is displayed in the picture box, as shown in Fig.C#-6.38:

Fig.C#-6.38

Let's now learn how to use the **SaveFileDialog** control.

Using the SaveFileDialog Control

The **SaveFileDialog** control is an instance of the **SaveFileDialog** class, which inherits the **FileDialog** class. It allows you to work with the **Save As** dialog box. This dialog box allows you to specify the name, type, and location to save a file. Using the **SaveFileDialog** control, you can either create a new file or overwrite an existing file.

Table 6.20 lists the noteworthy properties of the **SaveFileDialog** class:

Table 6.20: Noteworthy Properties of the SaveFileDialog Class	
Property	**Description**
`CreatePrompt`	Retrieves or sets a value that indicates whether the **Save As** dialog box prompts the user to create a new file if a non-existent file is specified
`OverwritePrompt`	Retrieves or sets a value specifying whether the **Save As** dialog box displays a warning if the user specifies the name of an existent file

Table 6.21 lists the noteworthy methods of the **SaveFileDialog** class:

Table 6.21: Noteworthy Methods of the SaveFileDialog Class	
Method	**Description**
`OpenFile`	Opens the selected file with read/write permission
`Reset`	Resets all dialog box options to their default values

Let's create a simple Windows Forms application named **SaveFileDialogDemo** to use the **Save As** dialog box. Perform the following steps to create :

1. *Repeat* steps **1-4** under the **Using the ToolStrip Control** heading of this chapter.
2. *Enter* **SaveFileDialogDemo** as the name of the application in the **Name** text box of the **New Project** dialog box. *Enter* the path of the folder, where you want to save the application, in the **Location** combo box.
3. *Click* the **OK** button in the **New Project** dialog box. The **SaveFileDialogDemo** application is created.
4. *Select* the **Form1** form in the designer and *press* the **F4** key to open the **Properties** window and *set* the **Text** property of the **Form1** form to **SaveFileDialogDemo**.
5. *Drag* **Label**, **Button**, and **TextBox** controls from **Toolbox** and *drop* them on the **Form1** form and set their properties to the values, given in Table 6.22:

Table 6.22: Values of the Properties of the Label, Button, and Text Box on Form1		
Control	**Property**	**Value**
`label1`	Text	Enter the file contents below:
	Location	12, 18
`button1`	Text	Save
	Location	107, 223
	Size	75, 23
`textBox1`	Multiline	True
	Location	15, 56
	Size	257, 134

6. *Drag* and *drop* the **SaveFileDialog** control on the **Form1** from the **Dialogs** tab of **Toolbox**, as shown in Fig.C#-6.39:

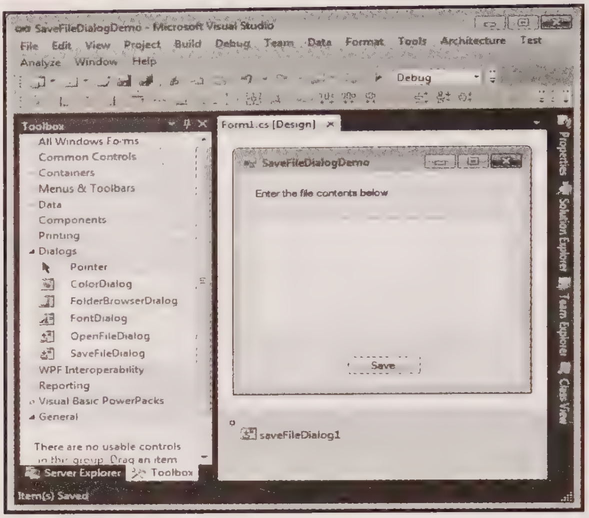


Fig.C#-6.39

7. *Double-click* the **Save** button and *add* the highlighted code, given in Listing 6.9, in its **Click** event:

Listing 6.9: Showing the Code of the button1_Click Event

```
private void button1_Click(object sender, EventArgs e)
{
   saveFileDialog1.Filter = "Text Files (*.txt*)|*.txt*";
   saveFileDialog1.InitialDirectory = Environment.SpecialFolder.MyDocuments.ToString();
   if (saveFileDialog1.ShowDialog() == DialogResult.OK)
   {
       if (saveFileDialog1.CheckFileExists == false)
         System.IO.File.WriteAllText(saveFileDialog1.FileName, textBox1.Text,
          System.Text.Encoding.UTF8);
    }
    textBox1.Text = "";
}
```

In Listing 6.9, the **Filter** property of the **saveFileDialog1** control is set such that only the text files are displayed in the **Save As** dialog box. In addition, the **InitialDirectory** property of the **saveFileDialog1** control is set to the **MyDocuments** folder located on the C drive. This implies that the initial location for saving the file is the **MyDocuments** folder. If a user clicks the **OK** button in the **Save As** dialog box, it is checked whether a file with the same name as that specified by the user exists on the computer. If such a file does not exist, then the text that the user entered in the **textBox1** control is saved in the specified file by using the **System.IO.File.WriteAllText()** method.

Note

The Filter and InitialDirectory properties are defined in the FileDialog class, which is inherited by the SaveFileDialog class.

8. *Press* the **F5** key to run the application.

9. *Enter* some text in the text box below the **Enter the file contents below** label (Fig.C#-6.40).

   In the subsequent steps, the text that you have entered in the text box is saved in a file by using the **Save As** dialog box.

10. *Click* the **Save** button, as shown in Fig.C#-6.40:

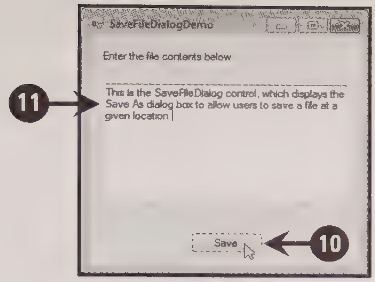


Fig.C#-6.40

The **Save As** dialog box appears.

11. *Specify* an appropriate location for the file in the location combo box located at the top of the **Save As** dialog box.
12. *Enter* the name with which you want to save the file in the **File name** text box. In this case, we have entered the file name as **MyFile** . Note that it is essential to specify the extension along with the name of the file.
13. *Click* the **Save** button in the dialog box to save the file.

Now, let's learn about the printing controls.

Using the Printing Controls

In C# 2010, there are several controls that allow you to incorporate the functionality of printing documents in .NET applications. There are two main controls that pertain to the printing process in C# 2010. They are as follows:

- **PrintDocument** control
- **PrintDialog** control

Now, let's learn about these two controls in the following section.

The PrintDocument Control

The **PrintDocument** control represents an object, containing the document that a user wants to print. This control allows you to set various properties, such as **DefaultpageSettings**, and **DocumentName** for printing a document. As the **PrintDocument** control does not have any UI, therefore, it is not visible at run time.

The **PrintDocument** control is an instance of the **PrintDocument** class that exists in the **System.Drawing.Printing** namespace. The **PrintDocument** class has several properties and methods that facilitate the printing process.

Table 6.23 lists noteworthy properties of the **PrintDocument** class:

Table 6.23: Noteworthy Properties of the PrintDocument Class

Property	Description
`DefaultPageSettings`	Retrieves or sets default page settings for all pages to be printed
`DocumentName`	Retrieves or sets the name of the document to display while printing the document
`PrinterSettings`	Retrieves or sets the printer that prints the document

Table 6.24 lists noteworthy events of the **PrintDocument** class:

Table 6.24: Noteworthy Events of the PrintDocument Class

Event	Description
`BeginPrint`	Occurs after the Print() method is called but before the first page is printed
`EndPrint`	Occurs after the last page of the document is printed
`PrintPage`	Occurs when output to print for the current page is needed
`QueryPageSettings`	Occurs just before a PrintPage event occurs

The **PrintDocument** class also has a method named **Print**, which allows you to start the printing process of a document.

The PrintDialog Control

The **PrintDialog** control displays the **Print** dialog box that allows you to print a document. The **Print** dialog box allows you to select a printer and specify various options and settings, such as whether you want to print all the pages or only specific pages and how many copies of the document are to be printed. Note that without the **PrintDialog** control; the document is printed using the default printer and printer settings.

The **PrintDialog** control is an instance of the **PrintDialog** class that exists in the **System.Windows.Forms** namespace. The **PrintDialog** class provides several properties and methods that you can use to print documents.

Table 6.25 lists noteworthy properties of the **PrintDialog** class:

Table 6.25: Noteworthy Properties of the PrintDialog Class

Property	Description
`AllowCurrentPage`	Retrieves or sets a value that indicates whether the users have the facility to print only the current page
`AllowPrintToFile`	Retrieves or sets a value that indicates whether the Print to file check box is enabled
`AllowSomePages`	Retrieves or sets a value that indicates whether the users have the facility to specify a range of pages
`Document`	Retrieves or sets the PrintDocument object
`PrinterSettings`	Retrieves or sets the printer settings which are modified in the Print dialog box
`PrintToFile`	Retrieves or sets a value that indicates whether the Print to file check box is selected

The **PrintDialog** class provides a method named **Reset** that allows you to reset all dialog box options to their default values.

Prior to displaying the **Print** dialog box using the **PrintDialog** control, you must set its **Document** property to a **PrintDocument** object. Moreover, you also need to set the **PrinterSettings** property of the **PrintDocument** object to the **PrinterSettings** property of the **PrintDialog** control.

Let's create a Windows Forms application named **PrintingDemo** to demonstrate the use of the **PrintDocument** and **PrintDialog** controls. Perform the following steps to create the **PrintingDemo** application:

1. *Repeat* steps **1-4** as under the **Using the ToolStrip Control** section of this chapter.
2. *Enter* **PrintingDemo** as the name of the application in the **Name** text box of the **New Project** dialog box. *Enter* the path of the folder where you want to save the application in the **Location** combo box. Now, *click* the **OK** button. The **PrintingDemo** application is created.
3. *Select* the **Form1** form in the designer and *press* the **F4** key to open the **Properties** window. *Set* the **Text** property of the **Form1** form to **PrintingDemo**.
4. *Drag* **Label**, **Button**, and **TextBox** controls from **Toolbox** and drop them on the **Form1** form in the designer. *Set* the properties of these controls to the values given in Table 6.26:

Table 6.26: Values of the Properties of the Label, Button, and Text Box Controls

Control	Property	Value
`label1`	Text	Enter the text to be printed:
	Location	25, 31
`button1`	Text	Print
	Location	101, 206

Table 6.26: Values of the Properties of the Label, Button, and Text Box Controls

Control	Property	Value
	Size	75, 23
textBox1	Multiline	True
	Location	28, 77
	Size	222, 91

5. *Drag* the **PrintDocument** and **PrintDialog** controls from the **Printing** tab of **Toolbox** and *drop* them on the **Form1** form, as shown in Fig.C#-6.41:

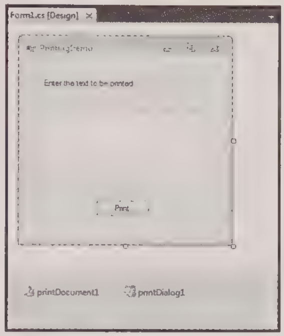


Fig.C#-6.41

As you can see in Fig.C#-6.41, both the **PrintDocument** and the **PrintDialog** controls appear in the component tray.

6. *Double-click* the **button1** button to *add* the highlighted code, as given in Listing 6.10, in its **Click** event handler in the **Form1.cs** file.

7. *Switch* to the designer of the **Form1** form and *double-click* the **printDocument1** control in the component tray. The **PrintPage** event handler of the **printDocument1** control is added to the **Form1.cs** file.

Listing 6.10 shows the highlighted code to be added for the **Click** event handler of the **button1** button and the **PrintPage** event handler of the **printDocument1** control:

Listing 6.10: Showing the Code of the button1_Click and printDocument1_PrintPage Event Handlers

```
private void button1_Click(object sender, EventArgs e) {
   printDocument1.DocumentName = "My Custom Document";
   printDialog1.Document = printDocument1;
   printDialog1.AllowSomePages = true;
   printDialog1.AllowCurrentPage = true;
   if (printDialog1.ShowDialog() == DialogResult.OK) {
         printDocument1.PrinterSettings = printDialog1.PrinterSettings;
         printDocument1.Print();
   }
}
private void printDocument1_PrintPage(object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{  Font textFont = new Font("Arial",14,FontStyle.Bold);
   e.Graphics.DrawString(textBox1.Text, textFont,Brushes.Black,new Rectangle(50,50,
         e.PageBounds.Height, e.PageBounds.Width));
}
```

In Listing 6.10, you see that in the **button1_Click** event handler, the **DocumentName** property of the **printDocument1** control is set to **My Custom Document**. This implies that **My Custom Document** is displayed as the name of document at the time of printing. In addition, the **Document** property of the **printDialog1** control is set to **printDocument1**, implying that the output to the printer selected in the **Print**

dialog box is sent by the **printDocument1** control. Both, the **AllowSomePages** and the **AllowCurrentPage** properties of the **printDialog1** control are set to **true**; which implies that the **Print** dialog box provides the users with the facility to print a sequence of pages as well as only the current page, respectively. When the **if** statement is true, that is, when the user clicks the **Print** button (equivalent to the **OK** button in other dialog boxes) in the **Print** dialog box, the printer settings specified in the **Print** dialog box are applied to the **printDocument1** control and the **Print()** method of the **printDocument1** control is called to start printing.

In the **printDocument1_PrintPage** event handler, various settings for printing the document are set. The **Font**, **Brushes**, and **Rectangle** classes of the **System.Drawing** namespace are used to print the document. Therefore, to use these three classes, you must include the **System.Drawing** namespace in your **Form1.cs** file through the **using** directive.

8. *Press* the **F5** key to run the application.
9. *Enter* some text in the text box below the **Enter the text to be printed** label (Fig.C#-6.42).
10. *Click* the **Print** button, as shown in Fig.C#-6.42:

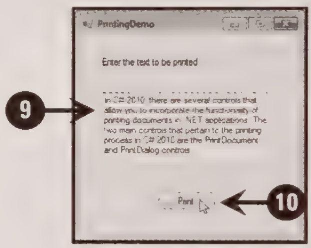


Fig.C#-6.42

The **Print** dialog box appears. In the **Print** dialog box, you can specify various options and preferences for printing the text. In this case, we have selected the **Current Page** radio button under the **Page Range** section and specified **2** as the **Number of copies** (Fig.C#-6.43).

11. *Click* the **Print** button in the **Print** dialog box to start printing the document, as shown in Fig.C#-6.43:

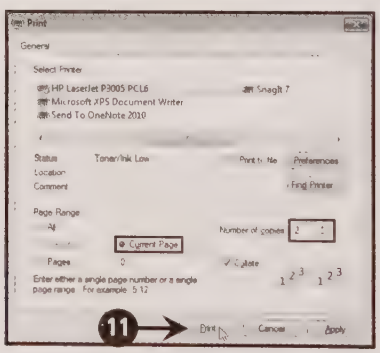


Fig.C#-6.43

Finally, let's recapitulate all the main points that you have learned in this chapter.

Summary

In this chapter, you have learned how to:

- Create toolbars, menus, and status bar
- Work with dialog boxes

Chapter 7

Introducing Windows Presentation Foundation and XAML

In this Chapter:

- Explaining the WPF 4.0 Architecture
- Exploring the Improvements in WPF 4.0
- Describing Types of WPF Applications
- Exploring the WPF 4.0 Designer
- Exploring XAML and WPF
- Working with WPF 4.0 Controls
- Working with Resources and Styles

Apart from including Windows Forms to develop desktop applications, Visual Studio 2010 encompasses Windows Presentation Foundation (WPF), previously known as Avalon, which offers several features and functionalities to develop high-end desktop applications. WPF supports various media, such as text, images, audio, and video, and allows you to work with two-dimensional (2D) as well as three-dimensional (3D) graphics. Unlike Windows Forms, which require numerous technologies, such as Graphics Device Interface (GDI+), Windows Media Player, and DirectX application programming interfaces (APIs) to work with 2D, 3D, and multimedia, WPF offers a cohesive medium that inherently provides the functionality of these technologies. This implies that you do not need to separate APIs in WPF to work with graphics, animations, and multimedia, and therefore it is easier and simpler to develop graphic-rich desktop applications by using WPF.

WPF was initially incorporated with .NET Framework 3.0 and was fully included in .NET Framework 3.5 and Visual Studio 2008. The version of WPF included in .NET Framework 4.0 is WPF 4.0.

In this chapter, you learn about the architecture of WPF 4.0, the improvements and enhancements in WPF 4.0, and the different types of WPF 4.0 applications. In addition, you learn about the WPF 4.0 Designer, the use of Extensible Application Markup Language (XAML) in WPF, and some of the common and new controls of WPF 4.0. You also learn how to use resources and styles in WPF.

We start the chapter by discussing the architecture of WPF 4.0.

Explaining the WPF 4.0 Architecture

Although WPF 4.0 is a part of .NET Framework 4.0, it has both managed and unmanaged components. The managed and unmanaged components of WPF 4.0 are shown in Fig.C#-7.1:

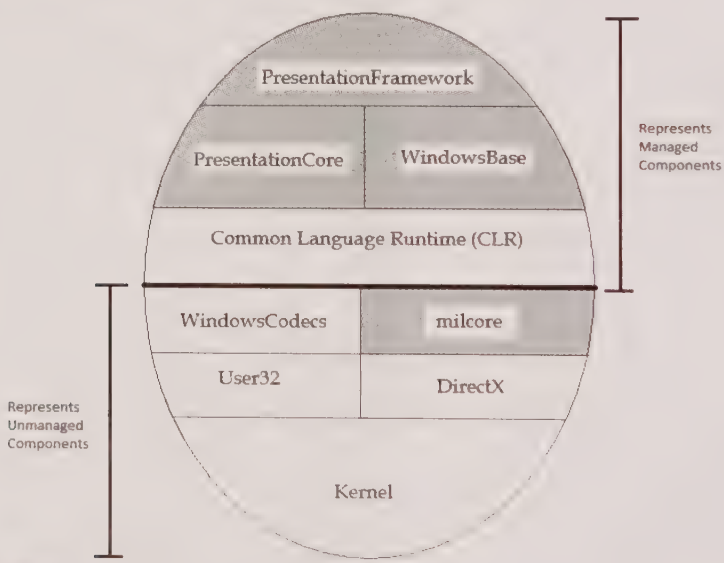


Fig.C#-7.1

As shown in Fig.C#-7.1, WPF 4.0 consists of the PresentationFramework, PresentationCore, WindowsBase, Common Language Runtime (CLR), Media Integration Layer (MIL or milcore), User32, DirectX, and Kernel components. The components that are shaded, that is, PresentationFramework, PresentationCore, WindowsBase, and milcore, are those that are essential to work with WPF 4.0 applications. To ensure that there is tight integration with DirectX, milcore is written in unmanaged code.

Note

The CLR, WindowsCodecs, User32, DirectX, and Kernel components of WPF 4.0 are part of the Microsoft Windows Vista operating system and it is beyond the scope of this chapter to discuss them.

Now, let's briefly discuss the components *of* WPF 4.0 in detail.

The PresentationFramework Component

The PresentationFramework component refers to the **PresentationFramework.dll** assembly in .NET Framework 4.0. This component offers classes to control the appearance and presentation of WPF 4.0 applications. For instance, controls, layout, and data binding in WPF applications are handled by the PresentationFramework component.

The PresentationCore Component

The PresentationCore component is implemented as the **PresentationCore.dll** assembly in .NET Framework 4.0. This component provides some of the most commonly used types and features of WPF 4.0. The classes and types offered by the PresentationCore component provide certain essential functionalities, such as properties, and events in WPF 4.0. Note that the PresentationCore component does not offer types for the user interface (UI) of WPF 4.0 applications such as those offered by the PresentationFramework component.

The WindowsBase Component

The WindowsBase component is implemented as the **WindowsBase.dll** assembly in .NET Framework 4.0. This component provides the fundamental features of WPF, such as threading. Some of these features can be accessed and used outside the WPF domain.

The MIL or Milcore Component

WPF 4.0 also contains an unmanaged component called milcore. An unmanaged component in WPF 4.0, milcore interacts with DirectX and acts as a medium or interface between CLR and DirectX (and managed WPF components). Due to this interaction with DirectX, milcore enables the display of 2D as well as 3D content in WPF 4.0 applications.

Now, let's discuss the various improvements in WPF 4.0.

Exploring the Improvements in WPF 4.0

WPF forms an essential component of the .NET Framework that enables .NET application developers to create WPF applications with visually appealing UIs. Using WPF 4.0, .NET developers can integrate 3D graphics, multimedia, animation, document support, speech recognition, and more into the applications. WPF 4.0 introduces a few improvements that include the following:

- **Addition of new controls:** Refers to the introduction of the new controls such as **DataGrid**, **Calendar**, and **DatePicker**. These controls enable application developers to create enhanced enterprise-level applications. In addition, these controls are 99% compatible with their Silverlight versions, which allow the reuse of code between Silverlight and WPF implementations.
- **Touch input and their manipulation:** Refers to the support for touch input that WPF 4.0 elements provide. The **UIElement**, **UIElement 3D**, and **ContentElement** classes consist of events that are raised when a user touches an element on a touch-screen. You can manipulate the **UIElement** class by rotating, scaling or translating it. For instance, when you can view some photographs in an application, you are able to zoom, move, resize, or rotate the photograph on the computer screen.
- **New text rendering stack:** Refers to the improvements made in rendering text, which include text clarity, configurability, and support for international languages. Moreover, users can now select from auto, aliased, grayscale or ClearType text rendering modes. In addition to all this, the text stack includes support for the display-optimized character layout so that text with sharpness comparable to Win32/GDI text can be generated.

- **Improved support for deployment:** Implies the improvements that have been made to the deployment size, time, and to the overall deployment experience. A WPF application can either target the standard installation of the full **.NET Framework 4.0** or **.NET Framework 4 Client Profile.** **.NET Framework 4 Client Profile** is a subset of the full **.NET Framework 4.0**. The overall application deployment experience is improved when WPF applications target **.NET Framework 4 Client Profile**, as it contains only those functionalities that are required by desktop applications, such as Windows Forms and WPF. Therefore, instead of targeting the full **.Net Framework 4.0**, desktop applications generally target **.NET Framework 4 Client Profile** by default.

Now that you have learned about the architecture of WPF 4.0 and the improvements made to it, let's discuss the type of WPF applications you can create.

Describing Types of WPF Applications

You can develop WPF 4.0 applications by using Visual Studio 2010. There are broadly two types of WPF applications supported by Visual Studio 2010, standalone WPF applications and XAML browser applications (also known as XBAPs). Let's learn about these two types of applications in detail, in the following sections.

Standalone WPF Applications

Standalone WPF applications are similar to Windows Forms applications, that is, you can install standalone applications on users' computers and view them from the **Start** menu. Note that a standalone WPF application has the same privileges as that of a currently logged-in user. In other words, this means that a standalone WPF application has full access to a system's resources, if the currently-logged on user has full access to the system's resources. Let's now learn how to create a standalone WPF application.

In Visual Studio 2010, you can easily and quickly create a standalone WPF application, by using the project template. When you use the project template of Visual Studio 2010 to create a standalone WPF application, the essential files are automatically added in the application.

Perform the following steps to create a standalone WPF application in Visual Studio 2010:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** on your computer to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar in the Visual Studio 2010 IDE, to open the **New Project** dialog box.
3. *Select* **Visual C#→Windows** in the **Installed Templates** pane of the **New Project** dialog box.

   The **New Project** dialog box contains the **WPF Application** project template in the middle pane. This template includes the initial files and folders required while creating standalone WPF applications.
4. *Select* **WPF Application** from the middle pane of the **New Project** dialog box.
5. *Enter* an appropriate name and location for the standalone WPF application in the **Name** text box and **Location** combo box, respectively. In our case, we have specified **MyFirstWPFApplication** as the name and selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects.**
6. *Click* the **OK** button in the **New Project** dialog box

   The New Project dialog box closes and a new standalone WPF application named **MyFirstWPFApplication** is created, as shown in Fig.C#-7.2:

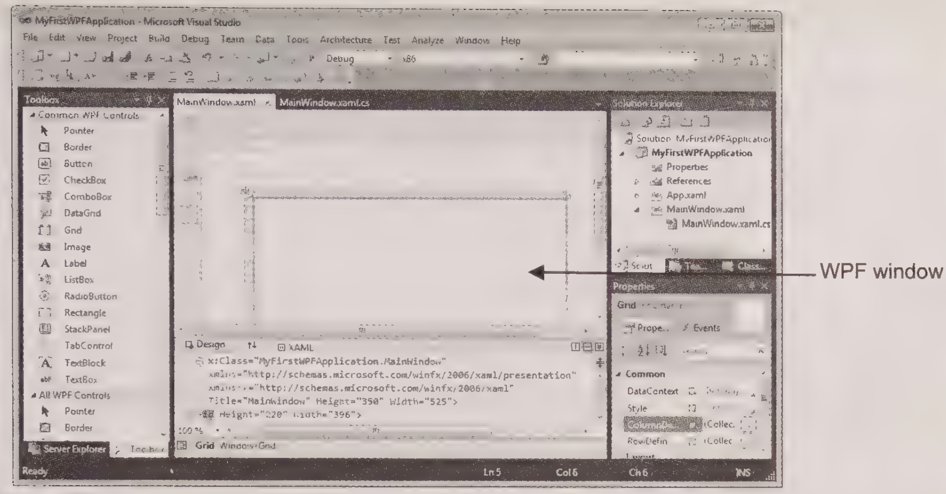


Fig.C#-7.2

As shown in Fig.C#-7.2, the designer interface of a standalone WPF application is similar to that of a Windows Forms application. For instance, tools are available in the Toolbox and a form-like structure called **MainWindow** can be seen at the center.

As stated earlier, using the **WPF Application** template for a standalone WPF application automatically generates the essential files and default UI for the application. Let's look at the different files that constitute a standalone WPF application. For this, open Solution Explorer, and expand the **References**, **App.xaml**, and **MainWindow.xaml** nodes, as shown in Fig.C#-7.3:

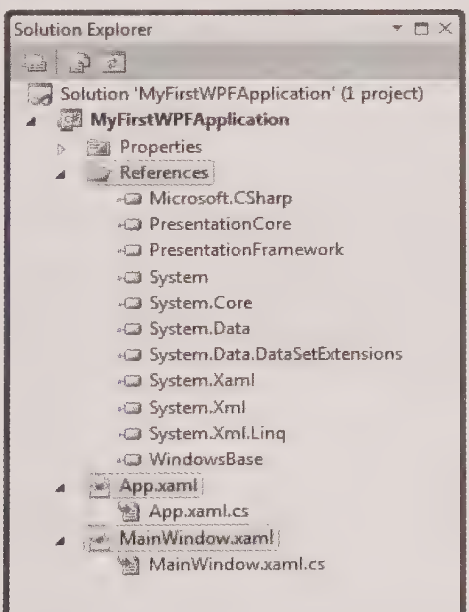


Fig.C#-7.3

Under the **References** node, you can view the essential .NET namespaces and assemblies, such as **PresentationCore, PresentationFramework**, and **WindowsBase**. When you expand the **App.xaml** node, you can view the respective code-behind file, **App.xaml.cs**. The **App.xaml** and **App.xaml.cs** files collectively represent the entire WPF application. The **App.xaml** file is an XAML file, while the **App.xaml.cs** file is the code-behind file (in C#) for the application. The **MainWindow.xaml** and **MainWindow.xaml.cs** files that are displayed in the Solution Explorer window collectively correspond to the default window of WPF 4.0 applications.

Note

In standalone WPF 4.0 applications, the content of an application is contained in one or more windows. A window is a rectangular region (similar to a Windows Form) and has a border, title, and buttons, such as Maximize, Minimize, and Close. WPF windows also have a menu that offers options to move and resize a window. You can see the Maximize, Minimize, and Close buttons and the menu only at run time, not at design time. WPF windows are instances of the Window class.

Let's now learn about XAML browser applications in WPF 4.0.

XAML Browser Applications

As the name suggests, XBAPs (pronounced as x-baps) are Web server-hosted WPF applications that run in a Web browser. XBAP applications consist of several pages, which users can navigate through in the browser. This is similar to the working of traditional websites. Unlike standalone WPF applications, XBAP applications are not installed on the computer of a user. This implies that if the user is offline (disconnected from the Internet or the server on which the XBAP application is hosted), then an XBAP application is inaccessible to the user.

In WPF 4.0, you can communicate with a Web page containing an XBAP application when it is hosted. In addition, based on the user's choice, an XBAP application can also be installed as a full trust application, which means that the application has access to the system's resources.

Note

To access XBAP applications, it is essential that the .NET Framework is installed on your computer.

Similar to standalone WPF applications, Visual Studio 2010 provides a project template to create an XBAP application in WPF 4.0.

Perform the following steps to create an XBAP application:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** on your computer to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar in the Visual Studio 2010 IDE, to open the **New Project** dialog box.
3. *Select* **Visual C#→Windows** in the **Installed Templates** pane of the **New Project** dialog box. The project templates for Visual C# Windows applications appear in the middle pane of the dialog box.
4. *Select* **WPF Browser Application** from the middle pane, to create an XBAP application.
5. *Specify* an appropriate name and location for the XBAP application in the **Name** text box and **Location** combo box, respectively. By default, the first XBAP application you create is named **WpfBrowserApplication1**; however, we have named our application **MyFirstWPFBrowserApplication**, and have selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects.**
6. *Click* the **OK** button in the **New Project** dialog box.

   The dialog box closes and the **MyFirstWPFBrowserApplication** application is created, as shown in Fig.C#-7.4:

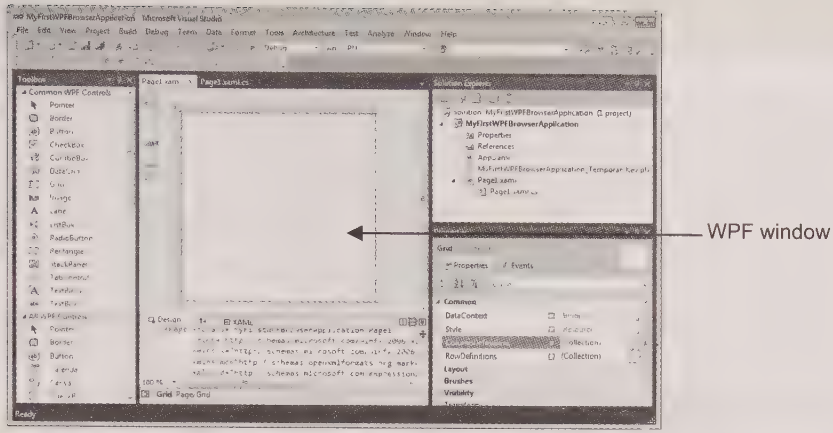


Fig.C#-7.4

As shown in Fig.C#-7.4, the UI of XBAP applications is similar to the UI of standalone WPF applications. In addition, some of the essential files, such as **App.xaml** and **App.xaml.cs**, are also part of the XBAP application. However, XBAP applications have the **Page1.xaml** and **Page1.xaml.cs** files instead of the **MainWindow.xaml** and **MainWindow.xaml.cs** files in standalone WPF applications. The **Page1.xaml** and **Page1.xaml.cs** files can be viewed in the Solution Explorer window and correspond to the **MainWindow.xaml** and **MainWindow.xaml.cs** files in a standalone WPF application.

Note

A WPF page is the basic entity in a XBAP application and is similar to a Web page. When you create an XBAP application, a page is automatically included in the application. You can also add new pages to the XBAP application. A single XBAP application page is an instance of the Page class.

Now let's explore the WPF 4.0 Designer in the next section.

Exploring the WPF 4.0 Designer

In Visual Studio 2010, the WPF 4.0 Designer is the designer interface that offers an easy, quick, and interactive way to work with the UI of WPF applications. Before you start working with full-fledged WPF applications, let's go through the important WPF Designer elements for a standalone WPF application, as shown in Fig.C#-7.5:

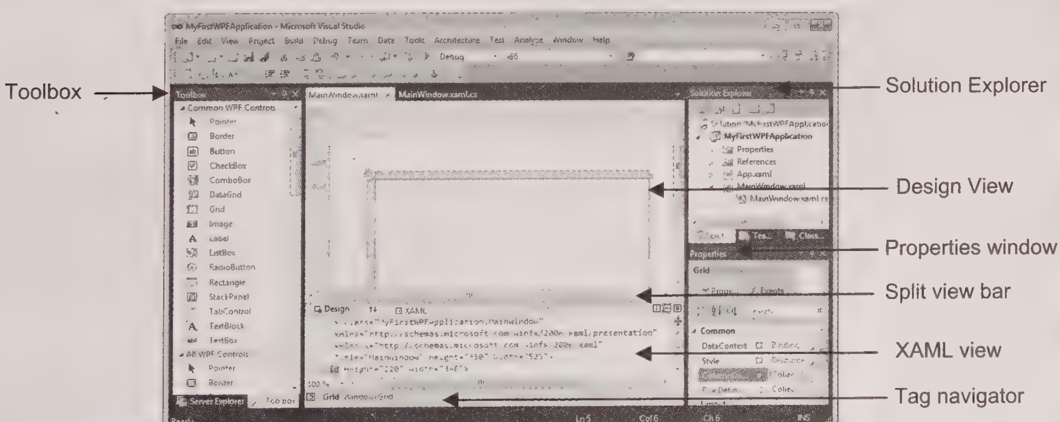


Fig.C#-7.5

As shown in Fig.C#-7.5, the WPF Designer resembles the Windows Forms Designer. For instance, the window, **MainWindow.xaml**, of the WPF Designer is quite similar to a form in a Windows Forms application. The **MainWindow.xaml** window constitutes the main design area of the WPF Designer just as a Windows Form constitutes the main design area of a Windows Forms Designer. In addition, both WPF Designer and Windows Forms Designer have Solution Explorer, the Properties window, and the Toolbox. The difference between the two is that the WPF Designer is divided into two primary views, the Design view and XAML view. Other designer elements, such as the split view bar and the tag navigator help you to work with the Design and XAML views.

Let's explore the Design and the XAML views, the split view bar, and the tag navigator in the following sections.

The Design View

As the name suggests, the Design view is the area where you build the visual aspects or UI of a WPF application by placing, dragging and resizing, and manipulating the appearance of the controls used in the application. The Design view allows you to easily and quickly build the UI of the WPF application in a What You See Is What you Get (WYSIWYG) manner, as shown in Fig.C#-7.6:

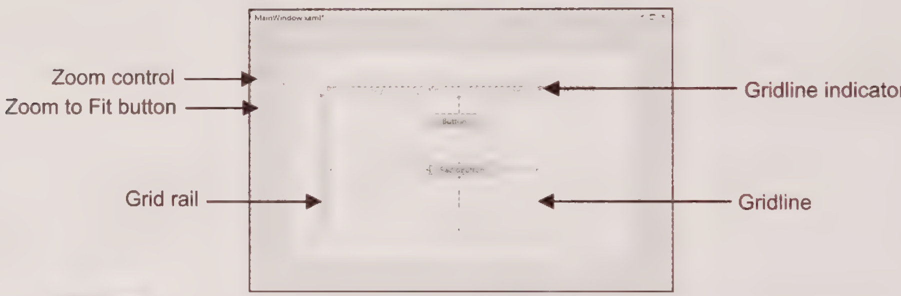


Fig.C#-7.6

As shown in Fig.C#-7.6, the Design view itself has several UI elements that help you to quickly design WPF applications. For instance, the Zoom control allows you to zoom in or out of the Design view; the move and resize handles allow you to appropriately position and resize the controls; and the margin lines allow you to set the margins of a control. Let's briefly discuss the elements of the Design view.

Grid Rails, Gridlines, and Gridline Indicators

The grid rails, gridlines, and gridline indicators pertain to the **Grid** control of WPF. When you create a WPF 4.0 application, a **Grid** control is added to the application by default. The **Grid** control allows you to represent a WPF application as a grid or lattice. The entire grid is enclosed by grid rails that span horizontally and vertically on the top and left respectively of the WPF application (Fig.C#-7.6).

By default, the grid has one row and one column; however, you can create additional rows and columns by clicking the desired positions on the grid rails. When you click the grid rails, gridlines and gridline indicators appear at those positions (Fig.C#-7.6). When you click the grid rail on the top, a vertical gridline appears, dividing the grid into two columns. Similarly, when you click the grid rail on the left, a horizontal gridline appears, dividing the grid into two rows. You can control the height and width of the rows and columns by moving the gridline indicators, which appear as triangles on the grid rails.

The Zoom Control and Zoom to Fit Button

You can find the Zoom control and the Zoom to Fit button at the upper-left corner of the Design view. The Zoom control allows you to zoom in and zoom out of the Design view. To zoom in or out, simply drag the

Zoom control slider up or down. The current zoom level, which in Fig.C#-7.7 is **100%**, can be seen on top of the Zoom control:

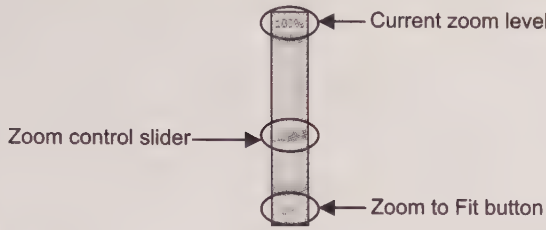


Fig.C#-7.7

In case you want to zoom or resize a window according to the available space in the WPF Design view, you can do so quickly by clicking the Zoom to Fit button, which appears just below the Zoom control.

The Move and Resize Handles

You can move and resize the controls in the Design view by using the move and resize handles, respectively. These handles appear only when you select a control. Note that the move handle appears only for some selected controls, such as the **Grid** control. For these controls, the move handle is located at the upper-left corner of the control (Fig.C#-7.8). You just need to click and drag the move handle to reposition the control in the Design view. To move those controls for which the move handle does not appear, you can simply select and drag them (Fig.C#-7.8).

When you select a control, you can also see its resize handles. The resize handles appear at the four corners of the control as square boxes, as shown in Fig.C#-7.8:

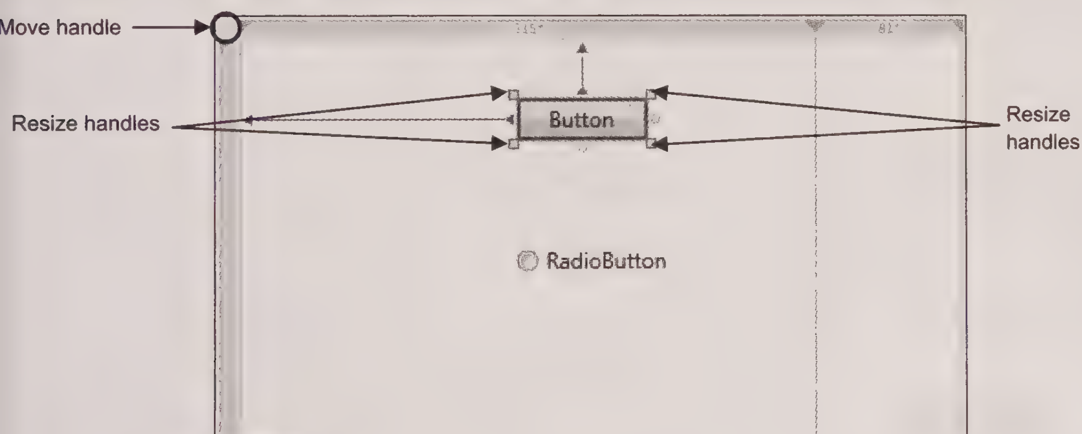


Fig.C#-7.8

To resize a control, click any of the handles and drag it outward or inward. Note that when you click the resize handles of a control, the current height and width of the control are displayed in maroon color on the right and bottom of the control, as shown in Fig.C#-7.9:

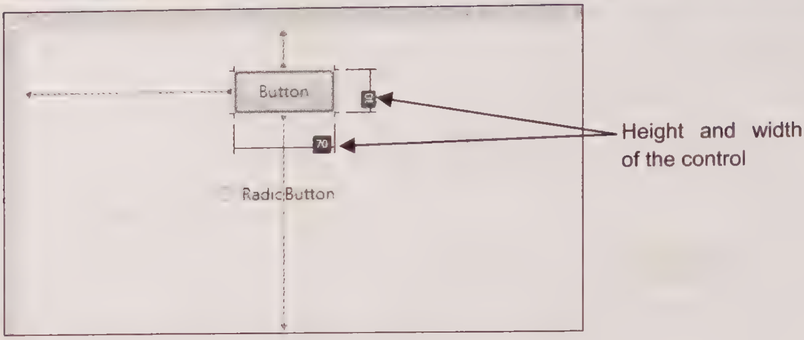


Fig.C#-7.9

Next, let's discuss about the margin lines and margin stubs in the design view of WPF applications.

Margin Lines and Margin Stubs

In WPF 4.0, controls exist within container controls. An example of a container control is the **Grid** control, which is automatically added to a WPF 4.0 application when you create the application. When you drag and drop the controls on the grid, you can change the distance, known as the *margin*, between a control and the grid. The margin is the distance from the edge of the control to the edge of its container control. If the container control is Grid, the margin is measured from the edge of the control to the nearest gridline. Each control has four margins: top, bottom, left, and right. These margins are represented by four lines, called margin lines, emerging from the edges of the control, as shown in Fig.C#-7.10:

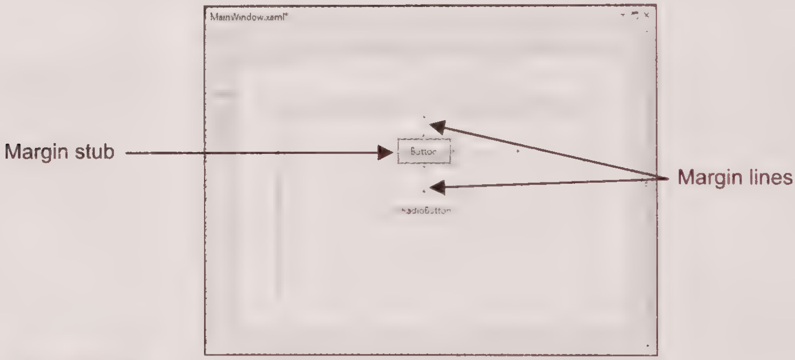


Fig.C#-7.10

Note that margin lines appear only when a control is selected.

You may notice in Fig.C#-7.10 that only the top, bottom, and right margin lines of the **Button** control appear; there is no left margin line. Instead, there is a small circle, called the *margin stub*, on the left edge of the **Button** control. The margin stub indicates that the respective margin is set to zero.

The margins for a control are collectively accessed through the **Margin** property of the control. You can set the **Margin** property of a control in the Design view, the XAML view, or the code-behind file. To set the **Margin** property of a control in the WPF Design view, select and drag the control. You can also use to set the **Margin** property by typing the values for the margins in the Properties window. You need to provide the values for the margins in a clockwise manner starting from the left margin, that is, left, top, right, and bottom. Note that when you type the new values for the margins in the Properties window, the size of the control may change.

Snaplines

Snapline is a feature in the WPF Design view that helps you to align the controls in a WPF 4.0 application. A snapline appears only when you have more than one control in a WPF application and you drag or align the edges and text of a control horizontally or vertically. The snapline appears as a light brown line along the edges of the control, as shown in Fig.C#-7.11:

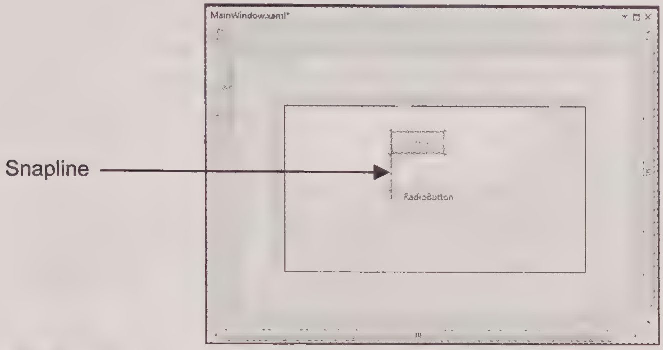


Fig.C#-7.11

In Fig.C#-7.11, you can see the number **49** displayed at one end of the snapline. This number represents the distance between the edges of the controls that you want to align. In case you do not want to see the snaplines, you can press the **ALT** key while dragging or resizing the control.

You are now familiar with the various UI components of the Design view. Let's move on to explore the XAML view of the WPF Designer.

The XAML View

As the name suggests, the XAML view allows you to view and work with the XAML code of WPF 4.0 applications. The XAML view and the Design view are inter-related such that whatever changes you make in the Design view are reflected in the XAML view and vice versa. This implies that you can also design WPF applications by using the XAML view. Fig.C#-7.12 shows the XAML view (along with the tag navigator) of a WPF 4.0 application:

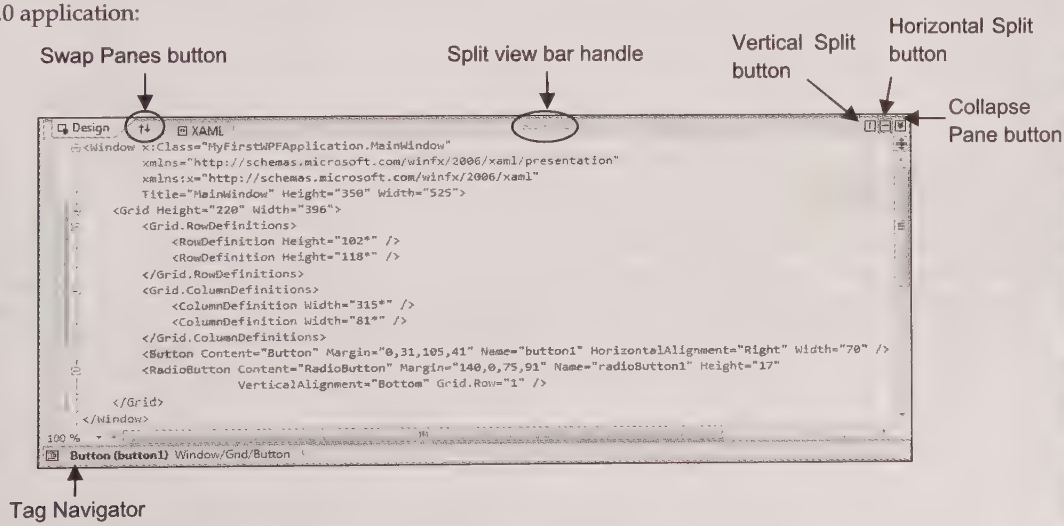


Fig.C#-7.12

Now let's discuss the split view bar.

The Split View Bar

In Visual Studio 2010, you can simultaneously work with the Design view and the XAML view. This is possible because of the split view bar (Fig.C#-7.12), which presents the split view of the WPF Designer. The split view refers to the separation or division of the WPF Designer into the Design view and XAML view. The split view bar has two tabs that correspond to the two panes on its left and the three buttons on its right. The first tab corresponds to the Design pane and it allows you to see the Design view, while the second tab corresponds to the XAML pane and it allows you to see the XAML view. There is a button, Swap Panes, between the Design and XAML panes on the split view bar. When you click the Swap Panes button, the Design view and XAML view are swapped with each other, as shown in Fig.C#-7.13:

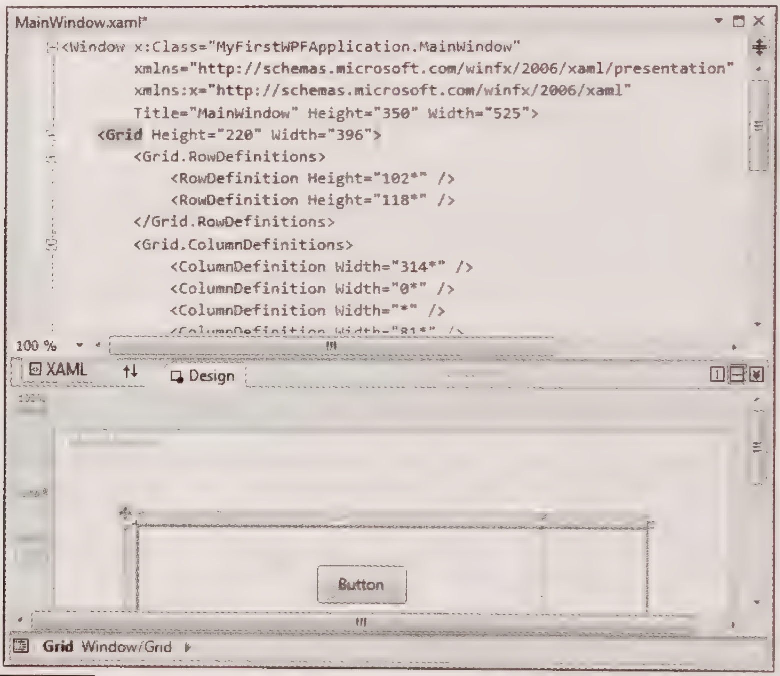


Fig.C#-7.13

You can also use the split view bar handle to resize the Design or XAML view. You can do so by dragging the split view bar handle upward or downward.

As stated earlier, there are three buttons on the right side of the split view bar. The first button, **Vertical Split**, allows you to vertically split the Design view and XAML view, as shown in Fig.C#-7.14:

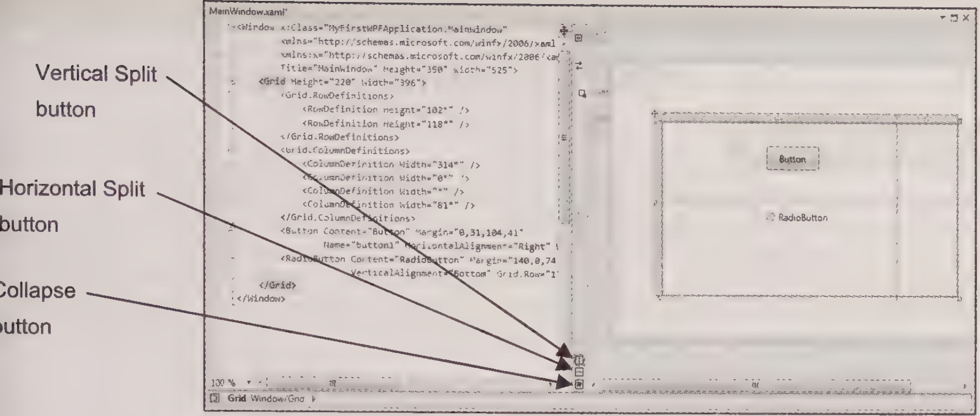


Fig.C#-7.14

The second button, Horizontal Split, allows you to see both the Design view and the XAML view horizontally, which is the default view. The third button, Collapse Pane, allows you to collapse or hide one of the panes. For example, when you click the Collapse Pane button, the Design pane collapses and you get more space for the XAML view. In case you want to expand the collapsed pane, you can click the Expand Pane button that appears in place of the Collapse Pane button.

The Tag Navigator

By default, the tag navigator appears just below the XAML view. It allows you to navigate to the parent XAML tag or element of the currently selected element. When you hover the mouse over a tag in the tag navigator, a thumbnail preview of that tag is displayed, as shown in Fig.C#-7.15:

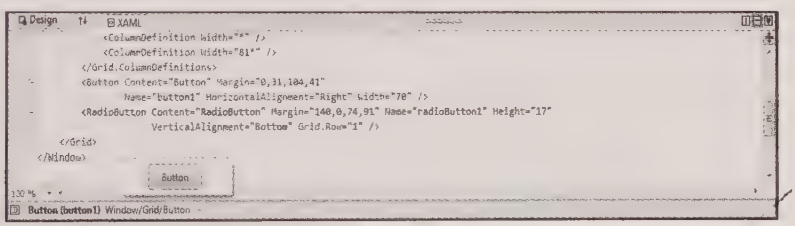


Fig.C#-7.15

As shown in Fig.C#-7.15, the **Button** control or tag is the currently selected tag in the XAML view. Its name is displayed in bold in the tag navigator. The name of the **Button** tag is followed by hyperlinks that are separated from each other by a forward slash, that is, **Window/Grid/Button**. The hyperlinks represent the hierarchy up to the currently selected tag. If you place the mouse pointer over any of the hyperlinks, a thumbnail image of the hyperlink is displayed (Fig.C#-7.16). You can click any of the hyperlinks to select that tag. Note that if the currently selected tag has any child tags, then the **Select Child** arrow is enabled and when you click the arrow, a list of the child tags appear, as shown in Fig.C#-7.16:

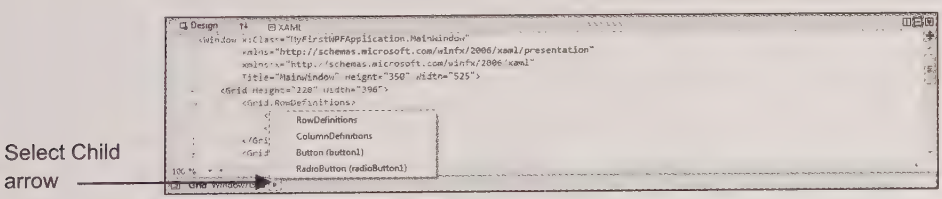


Fig.C#-7.16

This completes our discussion on the WPF Designer. Next let's discuss the XAML Designer in WPF 4.0.

Exploring XAML and WPF

WPF supports XAML (pronounced as *zammel*), which is a declarative markup language based on Extensible Markup Language (XML) introduced by Microsoft Corporation. This markup language allows designers to easily and quickly define and describe the elements for the UI of WPF applications. Application developers can then use the defined UI elements by using XAML to specify the logic for those elements in the code-behind files in any of the .NET languages, such as C#. However, you can also use the code-behind files to create UI elements at run time. Consequently, WPF allows both segregation and integration of the UI and business logic in WPF applications.

Note

XAML files have the .xaml extension, and are supported in Windows Workflow Foundation (WF) and Microsoft Silverlight.

Visual Studio 2010 extends support for XAML by incorporating XAML IntelliSense and allowing the debugging and compilation of XAML content. In the following sections, you learn to work with elements and attributes, namespaces, and markup extensions of XAML.

XAML Elements and Attributes

XAML makes use of markup tags or elements and attributes to define the UI of WPF applications. The XAML elements directly correspond to various managed WPF classes, while the XAML attributes correspond to the properties and events of the classes.

In WPF, the XAML elements are represented as a logical tree with several nodes. Each element corresponds to a tree node, while the attributes of the elements become the properties of the nodes. When you add an XAML element within an existing element, it becomes the child element of the existing element and therefore the child node of the existing node. In this way, as you keep adding XAML elements in a WPF application, the tree branches out to reflect the UI of the application. Note that in an XAML file, there is only one topmost or root element. In WPF standalone applications, the root element is the **MainWindow** element, while in XBAP applications; the root element is the **Page** element.

Let's create a simple WPF standalone application named **SampleWPFApplication.** Listing 7.1 shows the XAML code of the application:

Listing 7.1: Displaying the XAML Code of the **SampleWPFApplication** Application

```
<Window x:Class="SampleWPFApplication.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="MainWindow" Height="300" Width="300">
   <Grid>
         <Button Margin="100,25,100,200" Name="button1">Welcome</Button>
         <Button Margin="100,100,100,125" Name="button2">WPF</Button>
         <Button Margin="50,150,50,10" Name="button3">
                <Label Margin="5,5,5,5" Name="label1">XAML</Label>
         </Button>
   </Grid>
</Window>
```

As shown in Listing 7.1, there are four XAML elements, namely **Window, Grid, Button**, and **Label.** The **Window** and **Grid** elements exist in the WPF application by default, while the **Button** and **Label** elements are added by dragging and dropping the respective controls from the Toolbox. These elements can be represented in a hierarchical manner as a tree, as shown in Fig.C#-7.17:

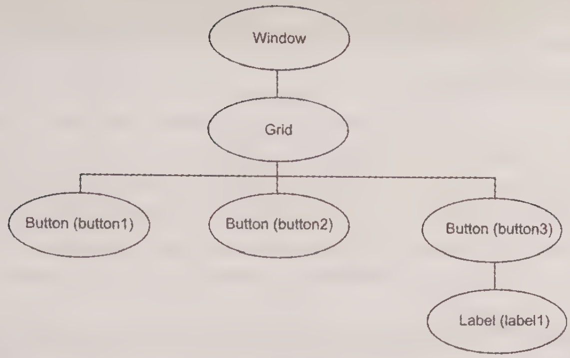


Fig.C#-7.17

In Fig.C#-7.17, the **Window** element contains the **Grid** element, which in turn has three **Button** elements named **button1**, **button2**, and **button3** as its child elements. The **button3** element itself has a **Label** element named **label1** as its child element. Note that the **Window, Grid, Button**, and **Label** XAML elements correspond to the instances of the **Window, Grid, Button**, and **Label** classes, respectively, while the attributes of the XAML elements correspond to the properties of the respective classes.

You may recall from Listing 7.1 that the individual attributes are separated by whitespaces and are specified in name-value pairs. For every attribute, the name and value of an attribute is separated by the equal to operator (**=**). Note that the value is enclosed within double quotes. Consider the following code snippet:

```
<Button Height="23" Name="button1">Welcome</Button>
```

In the preceding code snippet, the value of the **Height** attribute is **23** and the value of the **Name** attribute is **button1**. The string **Welcome** is the text that appears on the **button1** button.

Although setting the properties through XAML attributes is easy, concise, and intuitive, WPF provides an alternate method of setting the property values, known as the *property element syntax*. In this method, you can set the properties by specifying them as elements rather than as attributes. In the property element syntax, the property is set by using the following syntax:

```
<element_name.property_name [attribute1_name="value1"
  attribute2_name="value2"...attributeN_name="valueN"]>
property_value</element_name.property_name>
```

The various parameters of the preceding syntax can be briefly described as follows:

- `element_name`: Refers to the name of the element to which the property belongs
- `property_name`: Refers to the name of the property
- `attribute1_name, attrubute2_name, . . . attributeN_name`: Refer to the names of the attributes that the property may contain
- `value1, value2, . . .valueN`: Refer to the values of the attributes
- `property_value`: Refers to the value of the property

As shown in the preceding syntax, the dot operator (**.**) separates the element and property names. Listing 7.2 shows the property element syntax for the properties of the **Button** class/element:

Listing 7.2: Setting the Properties of the Button Element by using the Property Element Syntax

```
<Button>
   <Button.Height>23</Button.Height>
   <Button.Name>button1</Button.Name>
   <Button.Content>Welcome</Button.Content>
</Button>
```

In Listing 7.2, the **Height**, **Name**, and **Content** properties are set by using the property element syntax as the properties are specified as the child elements of the **Button** element.

Namespaces and XAML

Similar to XML, XAML requires certain predefined namespaces to allow you to use elements and attributes in an XBAP application. Namespaces convey to the XAML parser the location of the classes and structural rules for a WPF application. When you create a standalone WPF application, references to the necessary namespaces are added in the **Window** element and the **Page** element (in XBAP applications) by default.

By default, two namespaces are included in WPF applications, as shown in Listing 7.1. The namespaces are added by using the **xmlns** attribute of the **Window** or **Page** element. The **xmlns** attribute is set as shown in the following code snippet:

```
xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
```

In the preceding code snippet, the **xmlns** attribute is set to the default namespace for the XAML code and WPF applications. This namespace is provided as a Uniform Resource Indicator (URI). The namespace includes all the XAML elements and attributes that are required to work with WPF.

The second **xmlns** attribute is set as shown in the following code snippet:

```
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
```

In the preceding code snippet, the **xmlns** attribute is set to the namespace that is required to work with XAML in general. This namespace is more exhaustive than the default namespace as it provides all the essential XAML elements and attributes, including those that do not pertain to WPF, for example, the elements for WF and Silverlight.

Note that the second **xmlns** attribute (that refers to the XAML namespace) specifies **x:** as an alias for the namespace. This alias has to be prefixed with every element or attribute that belongs to the XAML namespace. For example, the **x:Class** attribute used in the **Window** element belongs to the XAML namespace and specifies the partial class (in the code-behind file) that implements the **Window** element, as shown in the following code snippet:

```
x:Class="WpfApplication1.MainWindow"
```

Note

You can add the xmlns attribute in any of the XAML elements. However, it is advisable to add the references in the root element of the XAML file, that is, the Window or Page element.

Markup Extensions

Till now you learned to set properties of elements, such as buttons, by using string or numeric values directly and explicitly. However, in some cases, you may not want to directly set the properties to string or numeric values; rather, you may want the properties to be set to a reference to a particular object. For such situations, WPF offers markup extensions. A markup extension is a means of setting the value of a property to an object reference. It also allows you to delay the assignment of a value to a property until run time or to bind a property to a value.

Markup extensions provide an easy, flexible, and efficient way of assigning values to properties through attributes as well as the property element syntax. In WPF, markup extensions are offered by various markup extension classes that help you to set the property to different types of values. The markup extension classes inherit the **System.Windows.Markup.MarkupExtension** class. Table 7.1 lists some of the common markup extension classes:

Table 7.1: Common Markup Extension Classes

Markup Extension Classes	XAML String Token	Description
BindingBase	Binding	Allows you to specify a data bound value for a given property
StaticResourceExtension	StaticResource	Allows you to specify an existing resource to determine the value of a property

Table 7.1: Common Markup Extension Classes		
Markup Extension Classes	**XAML String Token**	**Description**
DynamicResourceExtension	DynamicResource	Allows you to set a property value by delaying the reference to a given resource until run time

Markup extensions are characterized by their syntax, which is very different from the rest of the XAML content. You can provide a markup extension as either attributes or property elements. If you want to provide a markup extension as an attribute, you need to enclose it within the curly braces ({ }). The opening curly brace is followed by the string token for the markup extension class. The string token is followed by a whitespace, which is then followed by the input to the markup extension. Consider the following XAML code snippet:

```
<Button Margin="100" Name="button1" Content="Welcome" Style="{StaticResource
    mybackground}"/>
```

In the preceding code snippet, the **StaticResource** markup extension is used in the **Button** element to set the value for its **Style** property. This implies that the **Style** property of the **Button** element takes the value of the **mybackground** resource. You learn more about resources and styles later in this chapter.

Now, let's discuss the controls available in WPF 4.0.

Working with WPF 4.0 Controls

A control is a UI element that allows interaction between an application and the users of the application. WPF has a rich set of controls that allow you to develop high-end applications. Some of the WPF controls are similar to those in Windows Forms, such as the **Button** and **TextBox** controls. However, certain WPF controls, such as **Grid** and **Canvas,** are unique in terms of their appearance and behavior. In addition, some new controls, such as **DataGrid**, **Calendar**, and **DatePicker,** have been introduced in WPF 4.0.

In WPF, you can add controls and work with them in both the Design view and the code-behind file. However, it is recommended that you define the look of the controls in the Design view and the behavior of the controls in the code-behind file. You can add a control in a WPF application by dragging the control from the Toolbox and dropping it on the Design view. You can also add a control by double-clicking it in the Toolbox or writing the corresponding XAML code in the XAML view. Note that changes to a control in the Design view are automatically reflected in the XAML view and vice versa.

Note

Most of the WPF controls are derived from the System.Windows.Controls.Control class, which itself inherits the System.Windows.FrameworkElement class.

Some common controls available in WPF 4.0 are as follows:

- The **Grid** control
- The **Button** control
- The **TextBox** control
- The **PasswordBox** control
- The **TextBlock** control
- The **Border** control
- The **GridSplitter** control
- The **Canvas** control
- The **StackPanel** control
- The **DataGrid** control
- The **Calendar** control
- The **DatePicker** control

Let's discuss these controls one by one, starting with the **Grid** control.

Using the Grid Control

When you create a WPF application, you may notice that a **Grid** control is automatically added to the default window or page in the application. Every window or page in a WPF application (standalone or XBAP) contains a **Grid** control by default.

The **Grid** control is one of the most common and flexible container controls. A container control refers to a control that encompasses other controls and provides a predefined layout for these controls. The controls that are encompassed or contained within a container control are known as child controls or elements. In a **Grid** control, the child elements are contained in the cells of a grid. By default, a **Grid** control consists of a single cell, that is, one row and one column. However, you can create additional cells (rows and columns) in the **Grid** control by using the properties of the control.

The **Grid** control is an instance of the **Grid** class, which has various properties that allow you to work with the control. Table 7.2 lists the noteworthy properties of the **Grid** class:

Table 7.2: Noteworthy Properties of the Grid Class

Property	Description
ColumnDefinitions	Retrieves an ordered collection of columns defined in a Grid control
RowDefinitions	Retrieves an ordered collection of rows defined in a Grid control
ShowGridLines	Retrieves or sets a value that indicates whether or not the grid lines between the cells of a Grid control are visible
Column	Retrieves or sets a value that indicates the column in which a child element of a Grid control appears
ColumnSpan	Retrieves or sets a value that indicates the number of columns that a child element of a Grid control spans
Row	Retrieves or sets a value that indicates the row in which a child element appears in a Grid control
RowSpan	Retrieves the number of rows that a child element spans in a Grid control

Note

The Column, ColumnSpan, Row, and RowSpan properties listed in Table 7.2 are attached properties of the Grid control. An attached property is a property of a control that can be accessed and set by other controls. For instance, the Column, ColumnSpan, Row, and RowSpan properties of the Grid control can be accessed by its child elements.

Now, let's create a standalone WPF application named **GridDemo** to show the use of the **Grid** control. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar in the Visual Studio 2010 IDE, to open the **New Project** dialog box.
3. *Select* **Visual C#→Windows** in the **Installed Templates** pane and the **WPF Application** option from the middle pane of the **New Project** dialog box.
4. *Enter* the name **GridDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
5. *Click* the **OK** button. The **GridDemo** application is created.
6. Now *add* the highlighted code given in Listing 7.3, in the **MainWindow.xaml** file:

Listing 7.3: Adding Rows and Columns in a Grid Control

```
<Window x:Class="GridDemo.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Demo-Grid" Height="300" Width="300">
   <Grid ShowGridLines="True" Background="Cornsilk">
         <Grid.RowDefinitions>
                 <RowDefinition Height="68" />
                 <RowDefinition Height="68"/>
                 <RowDefinition Height="68"/>
                 <RowDefinition Height="68"/>
         </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
   <ColumnDefinition Width="139"/>
   <ColumnDefinition Width="139"/>
   </Grid.ColumnDefinitions>
   </Grid>
</Window>
```

In Listing 7.3, the **ShowGridLines** property of the **Grid** control is set to **True**, which implies that straight lines are visible between the rows and columns of the grid. The grid has two child elements, **Grid.RowDefinitions** and **Grid.ColumnDefinitions**. The **Grid.RowDefinitions** element refers to the **RowDefinitions** property of the grid, which allows you to define a collection of rows in the grid. The **Grid.ColumnDefinitions** element refers to the **ColumnDefinitions** property, which allows you to define a collection of columns in the grid. The **Grid.RowDefinitions** element has four **RowDefinition** elements, each corresponding to a row. Similarly, the **Grid.ColumnDefinitions** element has two **ColumnDefinition** elements, each of which corresponds to a column.

7 *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-7.18:

Fig.C#-7.18

Fig.C#-7.18 shows the grid lines between the rows and columns of a Grid control.

Using the Button Control

The **Button** control in WPF is similar to the one available for Windows Forms applications. It allows you to perform an action when a user clicks it. The **Button** control in WPF is a basic UI component that can contain text as well as an image.

The **Button** control is an instance of the **Button** class. Table 7.3 lists the noteworthy properties of the **Button** class:

Table 7.3: Noteworthy Properties of the Button Class

Property	Description
IsCancel	Retrieves or sets a value that specifies whether or not a Button control is a Cancel button
ClickMode	Retrieves or sets a value that indicates when the Click event of a Button control occurs, that is, when the mouse button is clicked, moved over the control, or released

Table 7.3: Noteworthy Properties of the Button Class

Property	Description
Content	Retrieves or sets the content of a Button control
IsDefault	Retrieves or sets a value that specifies whether or not a Button control is the default button
IsDefaulted	Retrieves a value that specifies whether or not a Button control is activated when a user presses the ENTER key from the keyboard

Now, let's create a standalone WPF application named **ButtonDemo** to show the use of a Button control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **ButtonDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **ButtonDemo** application is created.
4. *Add* the highlighted code given in Listing 7.4, in the **MainWindow.xaml** file:

Listing 7.4: Showing the Use of the Button Control

```
<Window x:Class="ButtonDemo.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Demo-Button" Height="200" Width="250">
   <Grid>
         <Button Name="button1" Height="25" Margin="0,0,0,100" Width="75" Content="1"
         Background="Black" Foreground="White"/>
         <Button Name="button2" Height="25" Margin="0,0,0,50" Width="100" Content="2"
         Background="Black" Foreground="White"/>
        <Button Name="button3" Height="25" Width="125" Content="3" Background="Black"
         Foreground="White"/>
   </Grid>
</Window>
```

In Listing 7.4, three **Button** controls (**button1**, **button2**, and **button3**) are added as child controls to the **Grid** control. The **Content** property of the three buttons are set to **1**, **2**, and **3**, respectively.

5. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-7.19:

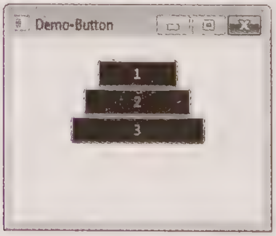


Fig.C#-7.19

In Fig.C#-7.19, the three button controls, button1, button2, and button3 are shown displaying the content 1, 2, and 3 in a Grid control.

Using the TextBox Control

If you want to allow users of a WPF application to enter textual data, you can use the **TextBox** control. This control provides the facility to enter or display unformatted text. With the **TextBox** control, you can manipulate

the input text by performing various operations, such as aligning the text, wrapping the text around the control, or specifying the casing of the text.

Note

By default, the TextBox control has a context menu, which appears when a user right-clicks in the control. The context menu allows the user to perform common editing operations such as Cut, Copy, and Paste.

The **TextBox** control is represented by the **TextBox** class, which is derived from the **TextBoxBase** class. The **TextBox** class has many properties that allow you to work with the **TextBox** control. Table 7.4 lists the noteworthy properties of the **TextBox** class:

Table 7.4: Noteworthy Properties of the TextBox Class

Property	Description
CharacterCasing	Retrieves or sets a value that indicates the casing of the characters of the text entered in a TextBox control
MaxLength	Retrieves or sets the maximum number of characters that you can enter in a TextBox control
Text	Retrieves or sets the text contained in a TextBox control
TextAlignment	Retrieves or sets the horizontal alignment of the text in a TextBox control
TextWrapping	Retrieves or sets a value that indicates how text is wrapped in a TextBox control

Let's create a standalone WPF application named **TextBoxDemo** to show the use of the **TextBox** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section in this chapter.
2. *Enter* the name **TextBoxDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **TextBoxDemo** application is created.
4. *Add* the highlighted code given in Listing 7.5, in the **MainWindow.xaml** file:

Listing 7.5: Showing the Use of the **TextBox** Control

```
<Window x:Class="TextBoxDemo.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Demo-TextBox" Height="250" Width="300">
  <Grid Background="Gray">
        <TextBox Name="textBox1" Height="75" Width="175" CharacterCasing="Upper"
        TextWrapping="Wrap" TextAlignment="Center" Background="LavenderBlush"
        VerticalScrollBarVisibility="Auto"/>
        <Button Height="25" Margin="100,150,100,0" Name="button1" Click="button1_Click"
        Background="LavenderBlush" Content="Clear TextBox"/>
  </Grid>
</Window>
```

In Listing 7.5, a **TextBox** control is added to a **Grid** control. The **TextBox** control is named **textBox1** and has its **CharacterCasing** property set to **Upper**, which implies that all the characters of the text that you enter appear in uppercase. The **TextWrapping** and **TextAlignment** properties of the **textBox1** control are set to **Wrap** and **Center,** respectively. This implies that the entered text is wrapped around the control and aligned at the center of the control. Note that another property, named **VerticalScrollBarVisibility,** of the **textBox1** control is set to **Auto**. This property is inherited from the **TextBoxBase** class and retrieves or sets a value that indicates whether a vertical scroll bar appears in the control.

5 *Double-click* **button1** in the Design view to open the code-behind file (**MainWindow.xaml.cs**) and *add* the **Click** event handler in the code-behind file.

6 *Add* the highlighted line of code in the **MainWindow** class, in the **Click** event handler of **button1**:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    textBox1.Text = "";
}
```

7 *Press* the **F5** key to run the application. The output appears (Fig.C#-7.20).

8 *Enter* some text in the **TextBox** control (Fig.C#-7.20).

9 *Click* the **Clear TextBox** button, as shown in Fig.C#-7.20:

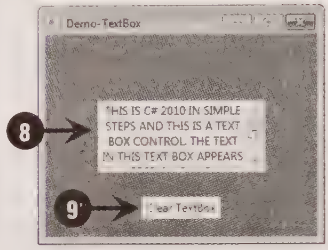


Fig.C#-7.20

When you click the **Clear TextBox** button, the text in the **TextBox** control is cleared, as shown in Fig.C#-7.21:

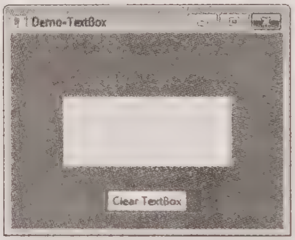


Fig.C#-7.21

Next, let's discuss about the PasswordBox control.

Using the PasswordBox Control

Sometimes, you want the users of your application to enter confidential information, such as passwords. You can use a **TextBox** control for this purpose. However, while entering the password, another user may accidentally see the password, which is an undesirable situation. A better alternative would be to use the **PasswordBox** control to enter passwords. The **PasswordBox** control allows you to obscure or mask the password while you enter it in the control. The **PasswordBox** control is a special type of text box in which the individual characters of the input text are masked with a given character so that the text appears as a string of that character. By default, any text entered in the **PasswordBox** control appears as a string of solid circles (●).

Note

The PasswordBox control uses a System.Security.SecureString object, which ensures that the text entered as a password in the control remains confidential.

The **PasswordBox** control is represented by the **PasswordBox** class, which has various properties that allow you to work with the **PasswordBox** control. Table 7.5 lists the noteworthy properties of the **PasswordBox** class:

Table 7.5: Noteworthy Properties of the PasswordBox Class	
Property	**Description**
MaxLength	Retrieves or sets the maximum length of the password in a **PasswordBox** control
Password	Retrieves or sets the password currently contained in a **PasswordBox** control
PasswordChar	Retrieves or sets the character to mask the password in a **PasswordBox** control

Now, let's create a standalone WPF application named **PasswordBoxDemo** to show the use of the **PasswordBox** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **PasswordBoxDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **PasswordBoxDemo** application is created.
4. *Add* the highlighted code given in Listing 7.6, in the **MainWindow.xaml** file:

Listing 7.6: Showing the Use of the PasswordBox Control

```
<Window x:Class="PasswordBoxDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Demo-PasswordBox" Height="225" Width="300">
        <Grid Background="Brown">
                <TextBox        Name="textBox1"        Height="20"         Width="123"
                        Margin="0,0,150,125"
                Text="Enter   your   User   ID"  IsReadOnly="True"   FontWeight="Bold"/>
        <TextBox    Name="textBox2"    Height="20"    Width="123"    Margin="0,0,150,30"
                Text="Enter your Password" IsReadOnly="True" FontWeight="Bold"/>
                <TextBox Name="textBox3" Height="20" Width="120"
                        Margin="150,0,0,125"
                Text="admin123"/>
        <PasswordBox      Name="passwordBox1"      Margin="150,0,0,30"       Width="120"
                Height="20" MaxLength="9" PasswordChar="*"/>
                <Button Name="button1" Height="23" Width="100"
                        Margin="0,100,0,0"
                Content="Login" Click="button1_Click" />
        </Grid>
</Window>
```

In Listing 7.6, a **PasswordBox** control, three **TextBox** controls, and a **Button** control are added to a **Grid** control. The **MaxLength** property of the **PasswordBox** control is set to **9**, which implies that a user can enter a maximum of nine characters in the **PasswordBox** control. The **PasswordChar** property is set to an asterisk (*), which refers to the mask that appears for each character entered in the **PasswordBox** control.

5. *Double-click* **button1** in the Design view. The event handler for the **Click** event is automatically added to the **MainWindow.xaml.cs** file.
6. *Add* the code highlighted given in Listing 7.7, in the **button1_Click** event handler:

Listing 7.7: Adding Code for the button1_Click Event

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    if (textBox3.Text == "" || passwordBox1.Password == "")
    MessageBox.Show("Please enter your user id and password");
    else if (!(textBox3.Text == "admin123" && passwordBox1.Password == "Nov2008"))
    MessageBox.Show("Invalid user id or password");
    else
    MessageBox.Show("You have successfuly logged in");
}
```

7 *Press* the **F5** key to run the **PasswordBoxDemo** application. The output appears (Fig.C#-7.22).

8 *Click* the **Login** button, as shown in Fig.C#-7.22:

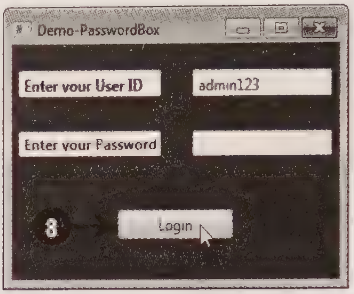


Fig.C#-7.22

A message box appears asking you to enter your user id and password (Fig.C#-7.23).

9 *Click* the **OK** button on the message box to close the message box, as shown in Fig.C#-7.23:

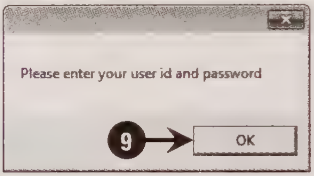


Fig.C#-7.23

The message box closes and the focus returns to the Demo-PasswordBox window.

10 *Enter* **Nov2008** in the password box.

11 *Click* the **Login** button.

A message box appears, stating that you have successfully logged in (Fig.C#-7.24).

12 *Click* the **OK** button in the message box, as shown in Fig.C#-7.24:

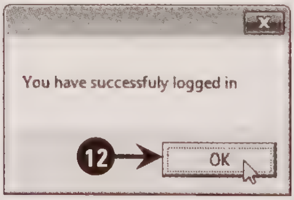


Fig.C#-7.24

This closes the message box.

Note

The TextBox and PasswordBox controls are simple controls, that is, controls that cannot have any child controls.

Next, let's learn how to use the **TextBlock** control.

Using the TextBlock Control

The **TextBlock** control in WPF allows you to work with text in a flexible manner as compared to the **Label** control. The appearance of the text that a **TextBlock** control contains can be easily manipulated by using several properties.

The **TextBlock** control is an instance of the **TextBlock** class, which offers various properties. Some noteworthy properties of the **TextBlock** class are listed in Table 7.6:

Table 7.6: Noteworthy Properties of the TextBlock Class

Property	Description
FontFamily	Retrieves or sets the font family for the text in a TextBlock control
FontSize	Retrieves or sets the font size for the text in a TextBlock control
FontStyle	Retrieves or sets the font style for the text in a TextBlock control
FontWeight	Retrieves or sets the font weight for the text in a TextBlock control
Text	Retrieves or sets the text of a TextBlock control
TextAlignment	Retrieves or sets a value that indicates the horizontal alignment of the text in a TextBlock control
TextTrimming	Retrieves or sets a value that indicates how text is trimmed when it overflows the content area of a TextBlock control
TextWrapping	Retrieves or sets a value that indicates how a TextBlock control should wrap the text within the control

Now, let's create a standalone WPF application named **TextBlockDemo** to show the use of the **TextBlock** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section in this chapter.
2. *Enter* the name **TextBlockDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **TextBlockDemo** application is created.
4. *Add* the highlighted code given in Listing 7.8, in the **MainWindow.xaml** file:

Listing 7.8: Displaying Text by Using the **TextBlock** Control

```
<Window x:Class=" TextBlockDemo.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Demo-TextBlock" Height="200" Width="300">
  <Grid >
       <TextBlock  Name="textBlock1" Height="150" Width="225" Text="WPF offers several
       features and functionalities to develop high-end desktop applications."
       Foreground="White" Background="Black" FontSize="17" FontWeight="Bold"
       TextWrapping="Wrap" TextAlignment="Center" />
  </Grid>
</Window>
```

In Listing 7.8, you can see that a **TextBlock** control is added to a **Grid** control. The **Text** property of the **TextBlock** control contains the text to be displayed on the control. The **Foreground** and **Background** properties of the control are set to **White** and **Black** respectively, which implies that the text in the **TextBlock** control appears in white, while the background appears black. The **FontSize** and **FontWeight** properties of the control are set to **17** and **Bold,** respectively so that the text appears large. In addition, the **TextWrapping** and **TextAlignment** properties are set to **Wrap** and **Center,** respectively. This implies that the text (if it spans multiple lines) is wrapped around the control and is displayed in the center of the control.

5. *Press* the **F5** key to run the **TextBlockDemo** application. The output appears, as shown in Fig.C#-7.25:

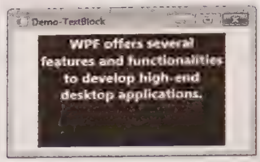


Fig.C#-7.25

In Fig.C#-7.25, the **TextBlock** control displays the input text wrapped around the control.

Using the Border Control

The **Border** control in WPF provides the facility to add a border, background, or both to a WPF control. It can have only one child element, which implies that at a time a **Border** control can apply a border or background to only a single element. In other words, you cannot apply the **Border** control to multiple child elements.

The **Border** control is an instance of the **Border** class, which has several properties. Some noteworthy properties of the **Border** class are listed in Table 7.7:

Table 7.7: Noteworthy Properties of the Border Class

Property	Description
Background	Retrieves or sets the background of a Border control
BorderBrush	Retrieves or sets the color of a Border control
BorderThickness	Retrieves or sets the relative thickness of a Border control
CornerRadius	Retrieves or sets the degree to which the corners of a Border control are rounded

Let's create a standalone WPF application named **GridWithBorder** to show the use of the **Border** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section in this chapter.
2. *Enter* the name **GridWithBorder** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **GridWithBorder** application is created.
4. *Add* the highlighted code given in Listing 7.9, in the **MainWindow.xaml** file:

Listing 7.9: Adding a Border to a Grid Control

```
<Window x:Class="GridWithBorder.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Grid with Border" Height="300" Width="300">
    <Grid>
    <Border Background="Pink" BorderBrush="Black" BorderThickness="9" CornerRadius="45"/>
    <Button Height="100" Width="150" Content="Example-Bordered Grid" FontWeight="Bold"/>
    </Grid>
</Window>
```

In Listing 7.9, a **Border** control is added as a child element to a **Grid** control in the **MainWindow.xaml** file. Through the **Background** property of the **Border** control, the background color of the **Grid** control is set to **Pink**. The **BorderBrush** and the **BorderThickness** properties of the **Border** control are set to **Black** and **9**, respectively. In addition, the **CornerRadius** property of the **Border** control is set to **45**, that is, the corners of the border are rounded to **45** degrees.

5. *Press* the **F5** key to run the **GridWithBorder** application. The output appears, as shown in Fig.C#-7.26:

Fig.C#-7.26

In Fig.C#-7.26, you can see that we have used a Border control to add a border to a Grid control.

Using the GridSplitter Control

The **GridSplitter** control in WPF offers a unique facility of redistributing the space between the cells (rows and columns) of a **Grid** control. With the **GridSplitter** control, the height and width of the **Grid** control do not change: only the space between the cells of the grid changes.

The **GridSplitter** control is an instance of the **GridSplitter** class, which has various properties to allow you to manipulate the spaces between the rows and columns of a **Grid** control. Table 7.8 lists the noteworthy properties of the **GridSplitter** class:

Table 7.8: Noteworthy Properties of the GridSplitter Class

Property	Description
PreviewStyle	Retrieves or sets the style to customize the appearance, effects, or other style characteristics for a GridSplitter control preview indicator that is displayed when the ShowsPreview property of the control is set to True
ResizeBehavior	Retrieves or sets which columns or rows are resized relative to the column or row for which a GridSplitter control is defined
ResizeDirection	Retrieves or sets a value that indicates whether or not a GridSplitter control resizes rows or columns
ShowsPreview	Retrieves or sets a value that indicates whether or not a GridSplitter control updates a column or row size as a user drags the control

Let's create a standalone WPF application named **GridSplitterDemo** to show the use of the **GridSplitter** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **GridSplitterDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **GridSplitterDemo** application is created.
4. *Add* the highlighted code given in Listing 7.10, in the **MainWindow.xaml** file:

Listing 7.10: Showing the Code to Split a Grid Control by Using a **GridSplitter** Control

```
<Window x:Class="GridSplitterDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Demo-GridSplitter" Height="300" Width="300">
    <Grid>
            <Grid.RowDefinitions>
            <RowDefinition Height="100"/>
            <RowDefinition Height="15"/>
            <RowDefinition Height="*"/>
```

```
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="48"/>
            <ColumnDefinition Width="96"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Button  Name="button1"  Grid.Row="0"  Grid.Column="0"  Background="Purple"
        Foreground="White" Content="Row1"/>
       <GridSplitter  Grid.Row="1"  Grid.ColumnSpan="3"  Height="100"  Width="245"
        ShowsPreview="True"  ResizeDirection="Rows"  ResizeBehavior="CurrentAndNext"
        HorizontalAlignment="Stretch" Background="Lavender" />
       <Button  Name="button2"  Grid.Row="2"  Grid.ColumnSpan="3"  Background="Navy"
        Foreground="White" Content="Row3"/>
    </Grid>
</Window>
```

In Listing 7.10, the **Grid** control is divided into nine cells: three rows and three columns. In the second row, a **GridSplitter** control is added. The **ShowsPreview** property of the **GridSplitter** control is set to **True**, which displays the preview of how the rows or columns or both are resized when the **GridSplitter** control is moved. The **ResizeDirection** and **ResizeBehavior** properties of the control are set to **Rows** and **CurrentAndNext**, respectively. This implies that the **GridSplitter** control can resize only the rows, specifically the current (second row) and the next row (third row).

5. *Press* the **F5** key to run the **GridSplitterDemo** application. The output appears.
6. *Drag* the **GridSplitter** control downwards, that is, toward the third row. The size of the second row increases, whereas the size of the third row decreases, as shown in Fig.C#-7.27:

Fig.C#-7.27

Next, let's discuss about the **Canvas** control.

Using the Canvas Control

The **Canvas** control is a layout control in WPF used to place some other controls (child controls). The child controls are positioned inside the **Canvas** control absolutely through the use of coordinates with respect to the dimensions of the **Canvas** control. Note that the **Canvas** control does not have its own layout but offers a flexible container for other controls.

The **Canvas** control is an instance of the **Canvas** class, which itself inherits the **Panel** class. The **Canvas** class has several properties that allow you to work with the **Canvas** control. The noteworthy properties of the **Canvas** class are listed in Table 7.9:

Table 7.9: Noteworthy Properties of the Canvas Class

Property	Description
Bottom	Retrieves or sets a value that indicates the distance between the bottom of a Canvas control and its child control
Left	Retrieves or sets a value that indicates the distance between the left borders of a Canvas control and its child control
Right	Retrieves or sets a value that indicates the distance between the right borders of a Canvas control and its child control
Top	Retrieves or sets a value that indicates the distance between the top of a Canvas control and the top of its child control

Note

The properties listed in Table 7.9 are attached properties.

Let's create a WPF standalone application named **CanvasDemo** to show the use the **Canvas** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **CanvasDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **CanvasDemo** application is created.
4. *Add* the highlighted code given in Listing 7.11, in the **MainWindow.xaml** file:

Listing 7.11: Showing the Use of the **Canvas** Control

```
<Window x:Class="CanvasDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Demo-Canvas" Height="300" Width="300">
    <Grid Background="Black">
            <Button Name="button1" Height="25" Width="250" Margin="0,200,0,0"/>
            <Button Name="button2" Height="25" Width="250" Margin="0,0,0,200"/>
            <Canvas Height="100" Width="200" Name="canvas1" Background="LightYellow" >
            <Button Name="button3" Width="50" Height="25" Canvas.Top="15"
                    Canvas.Left="15"
            Background="Black"/>
            <Button    Name="button4"    Width="50"    Height="25"    Canvas.Bottom="15"
            Canvas.Right="15" Background="Black"/>
            </Canvas>
    </Grid>
</Window>
```

In Listing 7.11, a **Canvas** control is added to a **Grid** control. The **Canvas** control has two buttons (**button3** and **button4**) as its child elements. The **button3** button appears at a distance of **15** units from both the top and left sides of the **Canvas** control. Similarly, the **button4** button appears at a distance of **15** units from both the bottom and right sides of the **Canvas** control.

5. *Press* the **F5** key to run the **CanvasDemo** application. The output appears, as shown in Fig.C#-7.28:

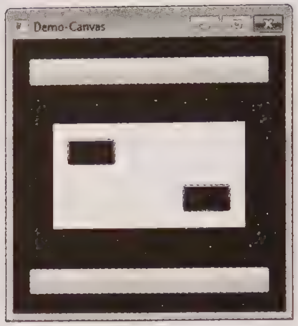


Fig.C#-7.28

In Fig.C#-7.28, you can see a Canvas control inside a Grid control containing the buttons button3 and button4 as child elements.

Using the StackPanel Control

Suppose you want to place multiple controls on a WPF window such that one appears on top of the other, forming a vertical stack of controls. In such a scenario, instead of using a **Grid** or **Canvas** control and explicitly placing these controls accordingly, you can use a **StackPanel** control. This control is another container control that allows you to position its child controls in a vertical or horizontal stack. By default, the child controls of the **StackPanel** control are stacked vertically. The vertical stack increases from top to bottom. You can also stack the child controls horizontally. The horizontal stack increases from left to right.

The **StackPanel** control is represented by the **StackPanel** class, which is derived from the **Panel** class. The **StackPanel** class has several properties that help you to work with the **StackPanel** control. Table 7.10 lists the noteworthy properties of the **StackPanel** class:

Table 7.10: Noteworthy Properties of the StackPanel Class

Property	Description
CanHorizontallyScroll	Retrieves or sets a value that indicates whether or not a StackPanel control can scroll horizontally
CanVerticallyScroll	Retrieves or sets a value that indicates whether or not a StackPanel control can scroll vertically
Orientation	Retrieves or sets a value that indicates whether the child controls of a StackPanel control are stacked horizontally or vertically

Let's create a standalone WPF application named **StackPanelDemo** to show the use the **StackPanel** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **StackPanelDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **StackPanelDemo** application is created.
4. *Add* the highlighted code given in Listing 7.12, in the **MainWindow.xaml** file:

Listing 7.12: Showing the Code to use a StackPanel Control

```
<Window x:Class="StackPanelDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    Title="Demo-StackPanel" Height="300" Width="300">
    <Grid>
    <StackPanel    Name="stackPanel1"    Height="75"    Width="125"    Margin="0,0,0,125"
          Orientation="Vertical" Background="SeaGreen">
                <Button Height="25" Name="button1" Width="75" Content="C#"/>
                <Button Height="25" Name="button2" Width="75" Content="VB"/>
                <Button Height="25" Name="button3" Width="75"
                        Content="ASP.NET"/>
          </StackPanel>
         <StackPanel   Name="stackPanel2"   Height="75"   Width="225"   Margin="0,125,0,0"
          Orientation="Horizontal" CanHorizontallyScroll="True" Background="SeaGreen">
          <Button Height="25" Name="button4" Width="75"  Content="C#"/>
             <Button Height="25" Name="button5" Width="75" Content="VB"/>
             <Button Height="25" Name="button6" Width="75" Content="ASP.NET"/>
          </StackPanel>
    </Grid>
</Window>
```

In Listing 7.12, the **Grid** control has two **StackPanel** controls as its child controls. Both the **StackPanel** controls have three buttons each as their respective child controls. However, the child controls of the **StackPanel** controls are stacked differently. In the first **StackPanel** control named **stackPanel1**, the three buttons (**button1**, **button2**, and **button3**) are stacked vertically, one below the other. This is because the **Orientation** property of the **stackPanel1** control is set to **Vertical**. However, the **Orientation** property of the second **StackPanel** control (named **stackPanel2**) is set to **Horizontal** and therefore has its child controls stacked horizontally.

5 *Press* the **F5** key to run the **StackPanelDemo** application. The output appears, as shown in Fig.C#-7.29:

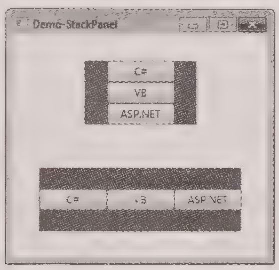


Fig.C#-7.29

In Fig.C#-7.29, you can see that the button controls in the StackPanel1 are aligned vertically and the button controls in the StackPanel2 control are aligned horizontally.

Using the DataGrid Control

DataGrid is a new control that has been introduced in WPF 4.0. Similar to the **DataGrid** control in ASP.NET Web Forms and Windows Forms, the **DataGrid** control in WPF 4.0 is used to display data in a customizable grid. The data inside a **DataGrid** control is displayed in a table, which is a collection of rows and columns. You can display the data from a database inside the **DataGrid** control or from any collection that implements the **IEnumerable** collection.

A **DataGrid** is an instance of the **DataGrid** class, which has various properties that allow you to work with the **DataGrid** control. Table 7.11 lists the noteworthy properties of the **DataGrid** class:

Table 7.11: Noteworthy Properties of the DataGrid Class

Property	Description
AlternatingRowBackground	Retrieves or sets the back color for alternating rows in a DataGrid control.
AutoGenerateColumns	Retrieves or sets a value indicating whether the columns are generated automatically in a DataGrid control.

Table 7.11: Noteworthy Properties of the DataGrid Class

Property	Description
CanUserAddRows	Retrieves or sets a value indicating whether or not a user can add new rows to a DataGrid control.
CanUserDeleteRows	Retrieves or sets a value that indicating whether or not a user can delete rows from a DataGrid control.
CellStyle	Retrieves or sets the style applied to all the cells in a DataGrid control.
Columns	Retrieves a collection that contains all the columns in a DataGrid control.
HasItems	Retrieves a value that indicates whether a DataGrid control contains any other controls, such as a TextBlock control or a Button control or items, such as strings within it.
Items	Retrieves a collection of all the controls and items that are used to populate a DataGrid control.
ItemTemplate	Retrieves or sets the DataTemplate objects used to display each item in a DataGrid control.
SelectedCells	Retrieves a list of cells that are currently selected in a DataGrid control.
SelectedIndex	Retrieves or assigns the index of the first item in the current selection or returns negative one (-1) if no item is selected in a DataGrid control.
SelectedItem	Retrieves or assigns the first item in the current selection or returns null if no item is selected in a DataGrid control.
SelectedValue	Retrieves or sets the value of the SelectedItem property in a DataGrid control.
Template	Retrieves or sets a control template in a DataGrid control.

Let's create a standalone WPF application named **DataGridDemo** to show the use of the DataGrid control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **DataGridDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **DataGridDemo** application is created.
4. *Add* the highlighted code given in Listing 7.13, in the **MainWindow.xaml** file:

Listing 7.13: Showing the use of a **DataGrid** Control

```
<Window x:Class="DataGridDemo.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataGrid-Demo" Height="350" Width="525">
    <Grid>
        <DataGrid AutoGenerateColumns="False" Height="200" HorizontalAlignment="Left"
         Margin="10,10,0,0" Name="dataGrid1" VerticalAlignment="Top"
         Width="265" RowBackground="Khaki"
          AlternatingRowBackground="Beige"></DataGrid>
    </Grid>
</Window>
```

In Listing 7.13, a **DataGrid** control is added to a Grid control. The **AutoGenerateColumns** property of the **DataGrid** control is set to **False**, which means that columns cannot be generated automatically in the

control. The **RowBackground** property of the **DataGrid** control is assigned the value **Khaki** while the **AlternatingRowBackground** property of the **DataGrid** control is assigned the value **Beige**, which implies that the background color of every alternate row in the **DataGrid** is beige.

The **DataGrid** we create in our application does not display any data.

5 *Add* the highlighted code shown in Listing 7.14 to the **MainWindow**.xaml.cs file to populate the **DataGrid** with data programmatically:

Listing 7.14: Showing the Code to Populate a **DataGrid** Control

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace DataGridDemo
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
            dataGrid1.Items.Add(new Student() { ID="ST01", Name = "John", Age = 14});
            dataGrid1.Items.Add(new Student() { ID="ST02", Name = "Andrew", Age = 11});
            dataGrid1.Items.Add(new Student() { ID="ST03", Name = "Katie", Age = 16});
            DataGridTextColumn col1 = new DataGridTextColumn();
            col1.Binding = new Binding("ID");
            DataGridTextColumn col2 = new DataGridTextColumn();
            col2.Binding = new Binding("Name");
            DataGridTextColumn col3 = new DataGridTextColumn();
            col3.Binding = new Binding("Age");
            dataGrid1.Columns.Add(col1);
            dataGrid1.Columns.Add(col2);
            dataGrid1.Columns.Add(col3);
            col1.Header = "Student_ID";
            col2.Header = "Name of Student";
            col3.Header = "Age of Student";
        }
        public class Student
        {
            private string name;
            private int age;
            private string id;
            public string Name
            {
                get;
                set;
            }
```

```
            public int Age
            {
                get;
                set;
            }
            public string ID
            {
                get;
                set;
            }
        }
    }
}
```

In Listing 7.14, a class named **Student**, is added to the **MainWindow.xaml** file of the **DataGridDemo** application.

The **Student** class contains three private data members, **id**, name and age. Three properties, **ID**, **Name** and **Age** are also defined to assign and retrieve the values of the **id**, **name**, and **age** data members defined in the **Student** class. Inside the constructor of the partial class **MainWindow**, the values of the **ID**, **Name**, and **Age** properties of the **Student** class are added to the **Items** collection of the **DataGrid** control, **dataGrid1**, by using the following lines of code:

```
dataGrid1.Items.Add(new Student() { ID="ST01", Name = "John", Age = 14});
dataGrid1.Items.Add(new Student() { ID="ST02", Name = "Andrew", Age = 11});
dataGrid1.Items.Add(new Student() { ID="ST03", Name = "Katie", Age = 16});
```

To display the values of the **ID, Name**, and **Age** properties inside the **dataGrid1** control, three columns, **col1**, **col2,** and **col3** are created and bound to the **ID**, **Name**, and Age properties, respectively, by using the following lines of code:

```
DataGridTextColumn col1 = new DataGridTextColumn();
col1.Binding = new Binding("ID");
DataGridTextColumn col2 = new DataGridTextColumn();
col2.Binding = new Binding("Name");
DataGridTextColumn col3 = new DataGridTextColumn();
col3.Binding = new Binding("Age");
```

Note that **DataGridTextColumn** is a class that is used to display textual content inside a **DataGrid** control. The **col1**, **col2**, and **col3** columns are then added to the **dataGrid1** control, by using the following lines of code:

```
dataGrid1.Columns.Add(col1);
dataGrid1.Columns.Add(col2);
dataGrid1.Columns.Add(col3);
```

The headers for **col1**, **col2**, and **col3** columns are set to clearly identify what data each of these columns is displaying in the **dataGrid1** control, by using the following lines of code:

```
col1.Header = "Student_ID";
col2.Header = "Name of Student";
col3.Header = "Age of Student";
```

8 *Press* the **F5** key to run the **DataGridDemo** application. The output appears, as shown in Fig.C#-7.30:

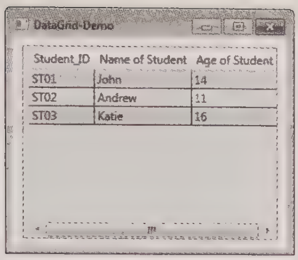


Fig.C#-7.30

In Fig.C#-7.30, data is displayed inside the **DataGrid** control under the **Student_ID**, **Name of Student**, and **Age of Student** columns. Next, let's discuss about the Calendar control.

Using the Calendar Control

The Calendar control is a new control that has been introduced with the latest version of WPF. It is a visual calendar, in which a user can select a particular date. Whenever a date is selected on a Calendar control, an event is fired on the selected date. The Calendar control is found in the System.Windows.Controls namespace, which is present in the PresentationFramework assembly. The Calendar control inherits the Control class. Table 7.12 lists some of important properties of the Control class:

Table 7.12: Noteworthy Properties of Calendar Control

Property	Description
BlackOutDates	Retrieves a collection of dates that are marked as not selectable in a Calendar control
DisplayDate	Retrieves or sets the date to display in a Calendar control
DisplayDateEnd	Retrieves or sets the last date in the date range that is available in a Calendar control
DisplayDateStart	Retrieves or sets the first date that is available in a Calendar control
DisplayMode	Retrieves or sets a value that indicates whether a Calendar control displays a month, year, or decade
FirstDayOfWeek	Retrieves or sets the day that is considered the beginning of the week in a Calendar control
SelectedDate	Retrieves or sets the currently selected date in a Calendar control
SelectedDates	Retrieves a collection of selected dates in a Calendar control
SelectionMode	Retrieves or sets a value that indicates what kind of selections are allowed in a Calendar control

Let's create a standalone WPF application named **CalendarDemo** to show the use the **Calendar** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **CalendarDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **CalendarDemo** application is created.
4. *Add* the highlighted code given in Listing 7.15, in the **MainWindow.xaml** file:

Listing 7.15: Using the Calendar Control

```
<Window x:Class="CalendarDemo.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="CalendarDemo" Height="300" Width="300">
```

```
    <Grid>
        <Calendar  Name="calendar1"  Height="170"  Width="185"  Margin="10,10,0,0"
        HorizontalAlignment="Left" VerticalAlignment="Top"/>
    </Grid>
</Window>
```

In Listing 7.15, a **Calendar** control is added to a Grid control. The Width and Height properties of the Calendar control are set while the Name property is used to uniquely identify the Calendar control.

5. *Press* the **F5** key to run the **CalendarDemo** application. The output appears, as shown in Fig.C#-7.31:

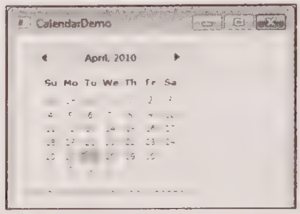


Fig.C#-7.31

In the Fig.C#-7.31, you see that a date is selected by default in the Calendar control. This is the current date or Today's date. You can change the **DisplayDate** property of the Calendar control to display whatever date you like.

The **DisplayMode** property of the Calendar control specifies the format of display for the Calendar, which can be month, year, or decade.

6. *Replace* the code in Listing 7.15 with the highlighted code given in Listing 7.16:

Listing 7.16: Showing the Code to Add in the **MainWindow.xaml** File

```
<Window x:Class="CalendarDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CalendarDemo" Height="300" Width="300">
     <Grid>
         <Calendar Name="calendar1" Height="170" Width="185" Margin="10,10,0,0"
          DisplayMode="Year" HorizontalAlignment="Left" VerticalAlignment="Top"/>
     </Grid>
</Window>
```

7. *Press* the **F5** key to run the **CalendarDemo** application again. The output appears, as shown in Fig.C#-7.32:

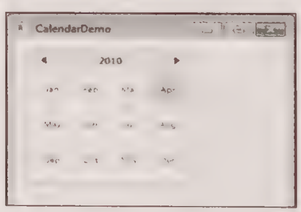


Fig.C#-7.32

8. Similarly, *replace* the code shown in Listing 7.16 with the highlighted code given in Listing 7.17 to limit the dates that you want to display in the Calendar control:

Listing 7.17: Showing the Code to Limit the Dates Displayed in a **Calendar** Control

```
<Window x:Class="CalendarDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CalendarDemo" Height="300" Width="300">
     <Grid>
         <Calendar DisplayMode="Year" DisplayDate="4/28/2010"
          DisplayDateStart="4/28/2010" DisplayDateEnd="4/30/2010"/>
     </Grid>
</Window>
```

9. *Press* the **F5** key to run the **CalendarDemo** application again. The output appears (Fig.C#-7.33).
10. *Click* the Apr option on the Calendar control, as shown in Fig.C#-7.33:

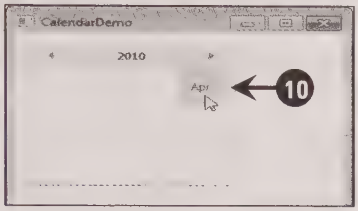


Fig.C#-7.33

The output on clicking the Apr option appears, as shown in Fig.C#-7.34:

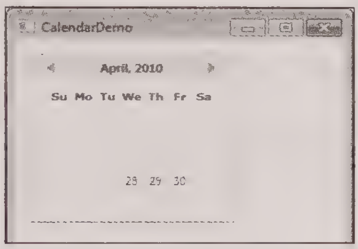


Fig.C#-7.34

In Fig.C#-7.34, we have limited the number of dates displayed in the **Calendar** control to three: **28**, **29**, and **30**.

Using the DatePicker Control

Similar to the Calendar control, a WPF DatePicker control also enables a user to select a date. Located in the System.Windows.Controls namespace, the DatePicker control inherits the DatePicker class.

Table 7.13 lists the noteworthy properties of the DatePicker class:

Table 7.13: Noteworthy Properties of DatePicker class

Property	Description
BlackOutDates	Retrieves or assigns a collection of dates that are marked as not selectable in a DatePicker control
DisplayDate	Retrieves or sets the date to display in a DatePicker control
DisplayDateEnd	Retrieves or sets the last date to be displayed in a DatePicker control
DisplayDateStart	Retrieves or sets the first date to be displayed in a DatePicker control
FirstDayOfWeek	Retrieves or sets the day that is considered the beginning of the week in a DatePicker control
SelectedDate	Retrieves or sets the currently selected date in a DatePicker control
SelectedDateFormat	Retrieves or sets the format used to display the selected date in a DatePicker control

Table 7.13: Noteworthy Properties of DatePicker class

Property	Description
SelectionMode	Retrieves or sets a value that indicates what kind of date selections are allowed in a DatePicker control. The default value for the SelectionMode property is SingleDate, which means that only one date can be selected.

Let's create a standalone WPF application named **DatePickerDemo** to show the use the **DatePicker** control. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **DatePickerDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **DatePickerDemo** application is created.
4. *Add* the highlighted code given in Listing 7.18, in the **MainWindow.xaml** file:

Listing 7.18: Showing the Code to Add a **DatePicker** Control

```
<Window x:Class="DatePickerDemo.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="DatePickerDemo" Height="300" Width="300">
   <Grid>
         <DatePicker Height="25" HorizontalAlignment="Left" Margin="10,10,0,0"
       Name="datePicker1" VerticalAlignment="Top" Width="115"/>
   </Grid>
</Window>
```

In Listing 7.18, a **DatePicker** control is added to a Grid control. The **Width** and **Height** properties of the **DatePicker** control are set while the **Name** property is used to uniquely identify the **DatePicker** control.

5. *Add* the highlighted code given in Listing 7.19 to the MainWindow.xaml.cs file:

Listing 7.19: Adding Code for the **DatePicker** Control

```
namespace DatePickerDemo
{
 /// <summary>
 /// Interaction logic for MainWindow.xaml
 /// </summary>
 public partial class MainWindow : Window
 {
  public MainWindow()
  {
   InitializeComponent();
   datePicker1.SelectedDate = new DateTime(2010, 5, 1);
   datePicker1.DisplayDateStart = new DateTime(2010, 5, 1);
   datePicker1.DisplayDateEnd = new DateTime(2010, 5, 31);
   datePicker1.FirstDayOfWeek = DayOfWeek.Monday;
  }
 }
}
```

As you can see in Listing 7.19, the DisplayDate property of the DatePicker control is set to May 1, 2010. The last date to be displayed on the DatePicker control is May 31, 2010, because of the DisplayDateEnd property. In addition, Monday is assigned as the first day of the week, using the FirstDayOfWeek property of the control.

6. *Press* the F5 key to run the **DatePickerDemo** application. The output appears (Fig.C#-7.35).
7. *Click* the down arrow on the right-hand side of the date displayed on the DatePicker control, as shown in Fig.C#-7.35:

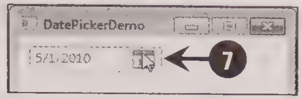


Fig.C#-7.35

The output on clicking the drop-down arrow is shown in Fig.C#-7.36:

Fig.C#-7.36

Now that you have learned about some of the common controls used in WPF, let's move ahead to learn about the resources and styles in WPF.

Working with Resources and Styles

Consider a situation where you want to apply a background color on several elements in a WPF application. For this, you may set the **Background** property on these elements; however, this may prove to be rather tedious. WPF allows you to simplify and ease the use of such commonly used objects through *resources.* A resource refers to an object, element, or value that is a part of the resource dictionary and is reusable by other elements of an application. The resource dictionary is an instance of the **ResourceDictionary** class and refers to a collection of resources defined on an element.

Although you can define resources for any UI element, it is advisable to define the resources on the root element, such as the **Window** or **Page** element. The resource that you define for an element also applies to all the child elements of the element. For example, if you define the resource for a **Window** element with a **Grid** element as its child, then the resource for the **Window** element applies to the **Grid** element also. However, if you define a resource for the **Grid** element, the resource applies only to the child elements of the **Grid** element.

Tip

Suppose you have a standalone WPF application with multiple Window elements and you want to apply a resource to all the Window elements. For this, you need to define the resource in the App.xaml file of the application.

You can define a resource on an element by using the **Resources** property of the element. The **Resources** property refers to the resource dictionary of that element. With the **Resources** property, you can define as many resources as you want on an element. Every resource that you define on an element must have a unique key, which refers to a string that uniquely identifies the resource in the resource dictionary. The unique key is assigned to a resource by using the **x:Key** attribute. After you define the resources, you can use them by referencing a resource as a property value in any of the UI elements. The syntax to reference a resource in WPF is as follows:

```
<elementName propertyName="{markupExtension keyName}">
```

```
<!-content-->
</elementName>
```

The various parameters in the preceding syntax can be briefly described as follows:

- **elementName**: Refers to the name of the element using the resource
- **propertyName**: Refers to the name of the property that takes its value from the resource
- **markupExtension**: Refers to the type of resource, which can be either a static resource or a dynamic resource
- **keyName**: Refers to the key name of the resource

Depending on the markup extension used to reference a resource, there are two types of resources, static and dynamic. Let's learn about these two resource types in detail in the following sections.

Using a Static Resource

Resources that are referenced by using the **StaticResource** markup extension are known as static resources. You can reference a resource as a static resource when the resource is static throughout run time, that is, the value of the resource does not change after it is referenced.

The value of a static resource is determined at the time of loading. The key name of the resource is looked up in all the resources (resource dictionaries) in an application. The key name is looked up first in the resources of the element for which you want to set the property. If the key name is not found, then the key name is looked up in the parent element and its resources up to the root element. If the key name is not found in any of the elements, the key name is looked up in the application resources. In case the static resource is not found at all, an exception occurs.

Let's create a standalone WPF application named **StaticResourcesDemo** to show the use of a static source. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **StaticResourcesDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **StaticResourcesDemo** application is created.
4. *Add* the highlighted code given in Listing 7.20, in the **MainWindow.xaml** file:

Listing 7.20: Using a Static Resource in XAML

```
<Window x:Class="StaticResourcesDemo.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Demo-StaticResources" Height="200" Width="300">
   <Grid>
         <Grid.Resources>
               <SolidColorBrush x:Key="firstBrush" Color="BlanchedAlmond"/>
               <SolidColorBrush x:Key="secondBrush" Color="ForestGreen"/>
         </Grid.Resources>
        <Button    Name="button1"    Width="50"    Height="100"    Margin="0,0,200,0"
         Background="{StaticResource firstBrush}"/>
        <Button    Name="button2"    Width="50"    Height="100"    Margin="0,0,0,0"
         Background="{StaticResource secondBrush}"/>
        <Button    Name="button3"    Width="50"    Height="100"    Margin="200,0,0,0"
         Background="{StaticResource firstBrush}"/>
   </Grid>
</Window>
```

In Listing 7.20, the **Grid** control uses the **Resources** property to define a set of resources. Two **SolidColorBrush** resources (**SolidColorBrush** elements that fill a given area with a solid color) with the

keys **firstBrush** and **secondBrush** are defined. These two resources are used to set the **Background** property of three buttons (**button1**, **button2**, and **button3**). The **StaticResource** markup extension is used to set the **Background** property to the resource values.

5. *Press* the **F5** key to run the **StaticResourcesDemo** application. The output appears, as shown in Fig.C#-7.37:

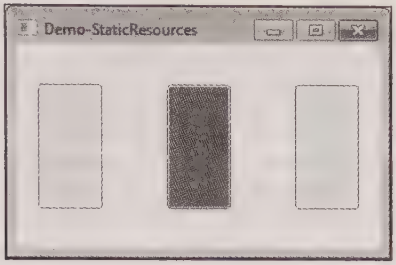


Fig.C#-7.37

In Fig.C#-7.37, the three buttons, **button1**, **button2**, and **button3** are displayed whose **Background** property is set using a **StaticResource** markup extension.

Let's now learn about dynamic resources.

Using a Dynamic Resource

When you use the **DynamicResource** markup extension, the resource is referred to as a dynamic resource. Compared to static resources, dynamic resources provide more flexibility and are useful in situations when you want to defer the assignment of a property value until run time, for instance, when you want to change the value of the resource at run time through user or system settings.

Unlike static resources that are looked up at the time of loading, dynamic resources are looked up only when they are used at run time. With dynamic resources, an expression is created for the requested resource. This expression is not evaluated until run time. When a dynamic resource is used at run time, the key of the requested resource is looked up in the resources of the element on which the property is being set. If the key is not found, then that key is looked up in the parent element and its resource dictionary up to the root element. If the key is still not found, then the application, theme, and system resources are looked up, in that order.

Let's create a standalone WPF application named **DynamicResourcesDemo** to show the use of dynamic resources. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.
2. *Enter* the name **DynamicResourcesDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **DynamicResourcesDemo** application is created.
4. *Add* the highlighted code given in Listing 7.21, in the **MainWindow.xaml** file:

Listing 7.21: Using a Dynamic Resource in XAML

```
<Window x:Class="DynamicResourcesDemo.MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Demo-DynamicResources" Height="300" Width="300">
   <Window.Resources>
         <SolidColorBrush x:Key="firstBrush" Color="Lavender"/>
         <SolidColorBrush x:Key="secondBrush" Color="PowderBlue"/>
         <SolidColorBrush x:Key="thirdBrush" Color="Violet"/>
         </Window.Resources>
         <Grid Name="grid1" Background="{StaticResource firstBrush}">
```

```
        <Button     Name="button1"     Height="100"     Width="150"     Margin="0,0,0,75"
         Content="Dynamic      Button"      BorderBrush="Black"      BorderThickness="5"
         Background="{DynamicResource grid1.Background}"/>
        <Button    Name="button2"    Height="25"    Margin="0,185,100,0"       Width="75"
         Content="Color1" Click="button2_Click" />
        <Button    Name="button3"    Height="25"    Margin="100,185,0,0"       Width="75"
         Content="Color2" Click="button3_Click" />
    </Grid>
</Window>
```

In Listing 7.21, you can see that the **Window** control has three **SolidColorBrush** resources with the keys **firstBrush**, **secondBrush**, and **thirdBrush**. The **firstBrush** resource is referenced in the **Background** property of the **grid1** control as a static resource. The **Background** property of the **grid1** control is then used as a dynamic resource in the **button1** control. This implies that whenever the background of the **grid1** control changes, the background of the **button1** control also changes.

5. *Double-click* the **Color1** button in the Design view to add the button's **Click** event handler in the **MainWindow.xaml.cs** file.

6. Now, *add* the following code snippet for the **button2_Click** event handler in the **MainWindow** class:

```
private void button2_Click(object sender, RoutedEventArgs e)
{
    grid1.Background = (Brush)this.FindResource("secondBrush");
}
```

In the preceding code snippet, in the **button2_Click** event handler, the background of the **grid1** control is set to the value of the **secondBrush** resource by using the **FindResource()** method.

7. *Switch* to the **MainWindow.xaml** file and *double-click* the **Color2** button in the Design view to add the button's **Click** event handler in the **MainWindow.xaml.cs** file.

8. *Add* the following code snippet for the **button3_Click** event handler in the **MainWindow** class:

```
private void button3_Click(object sender, RoutedEventArgs e)
{
    grid1.Background = (Brush)this.FindResource("thirdBrush");
}
```

In the preceding code snippet, the background of the **grid1** control is set to the value of the **thirdBrush** resource in the **button3_Click** event handler.

9. *Press* the **F5** key to run the **DynamicResourcesDemo** application. The output appears, as shown in Fig.C#-7.38:

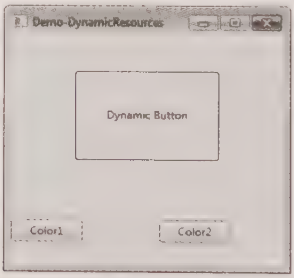


Fig.C#-7.38

10. *Click* the **Color1** button in the **Demo-DynamicResources** window, as shown in Fig.C#-7.39:

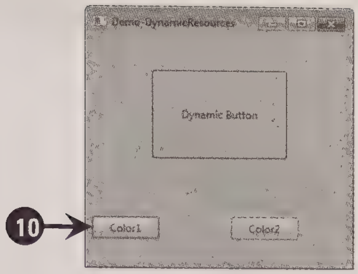


Fig.C#-7.39

Similarly, the background color of the Grid control changes when you click the button **Color2**, as shown in Fig.C#-7.40:

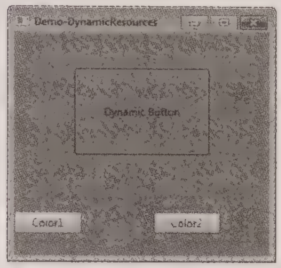


Fig.C#-7.40

In Fig.C#-7.40, we have used a dynamic resource to change the background color of the **Grid** control when the **Color2** button is clicked in the **Demo-DynamicResources** window.

Let's now learn how to set styles through resources.

Setting Style Through a Resource

One of the most common uses of resources in WPF is to apply styles uniformly to several elements. In most cases, styles are defined as resources in WPF applications and therefore are included in the resource dictionary of an element. In this way, styles become reusable entities, allowing them to be used with multiple elements in an application.

In WPF, you can use the Style element while defining the resources for an element. Each Style element has one or more Setter elements that specify the property and value to which the style is to be applied. The syntax of the Style element is as follows:

```
<Style x:Key="name" TargetType="elementName">
   <Setter Property="propertyName1" Value="propertyValue1"/>
   <Setter Property="propertyName2" Value="propertyValue2"/>
   .
   .
   <Setter Property="propertyNameK" Value="propertyValueK"/>
</Style>
```

The various parameters in the preceding syntax can be briefly described as follows:

- **x:Key**: Refers to the key name of the **Style** element
- **TargetType**: Refers to the name of the type of an element on which the style is applied
- **propertyName1**, **propertyName2**, . . . , **propertyNameK**: Refer to the names of the properties
- **propertyValue1**, **propertyValue2**, . . . , **propertyValueK**: Refer to the values of the properties

Let's create a standalone WPF application to show the use of styles as resources. For this, perform the following steps:

1. *Repeat* steps **1 to 3** as followed earlier while creating the **GridDemo** application in the *Using the Grid Control* section of this chapter.

2. *Enter* the name **StylesDemo** in the **Name** text box and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **StylesDemo** application is created.
4. *Add* the highlighted code given in Listing 7.22, in the **MainWindow.xaml** file:

Listing 7.22: Using Styles as Resources

```
<Window x:Class="StylesDemo.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Demo-Styles" Height="200" Width="300">
    <Grid>
        <Grid.Resources>
            <Style x:Key="myButtonStyle">
                <Setter Property="Button.Background" Value="Black"/>
                <Setter Property="Button.Foreground" Value="White"/>
            </Style>
        </Grid.Resources>
        <Button       Name="button1"      Style="{StaticResource      myButtonStyle}"
        Margin="10,10,140,85">Button1</Button>
            <Button      Name="button2"     Style="{StaticResource     myButtonStyle}"
        Margin="140,85,10,10">Button2</Button>
    </Grid>
</Window>
```

In Listing 7.22, the **Resource** property of the **Grid** control has a **Style** element with the key **myButtonStyle**. The **Style** element has two **Setter** elements that specify the values for the **Background** and **Foreground** properties of the buttons in the grid. The **Style** element is then referred to as a static resource in the **Style** property of the **button1** and **button2** controls. This implies that both the buttons in the grid have the same style.

5. *Press* the **F5** key to run the **StylesDemo** application. The output appears, as shown in Fig.C#-7.41:

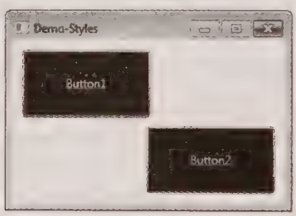


Fig.C#-7.41

With this, we come to the end of the chapter. Let's now recap the main points discussed in the chapter.

Summary

In this chapter, you have learned about:

- Architecture of WPF 4.0
- Enhancements in WPF 4.0
- Different types of WPF 4.0 applications
- WPF 4.0 Designer
- The use of XAML in WPF
- Using common and new controls WPF applications
- Using resources and styles in WPF applications

Chapter 8

ADO.NET and Data Binding

In this Chapter:

- Exploring ADO.NET
- Types of Data Binding in Windows Forms
- Data Binding in WPF

The world of business is growing rapidly and so is the need to store data. Data, as you know, is a collection of facts, which is generally stored in a database in the form of tables. A database is a collection of tables, which are used to store large amounts of data (information) systematically in a computer. This stored information can be accessed from the database quickly and efficiently as and when required. Databases that are used to relate data in multiple tables are called relational databases. Some popular relational databases are SQL Server (where SQL stands for Structured Query Language Server), Oracle, and Microsoft Access.

Retrieving and manipulating data directly from a database requires the knowledge of basic SQL commands. Anyone who is not familiar with SQL commands will not be able to use the data stored in a database. Therefore, most business applications provide a user-friendly interface that helps to retrieve data from a database without having to write SQL commands. Microsoft ActiveX Data Objects.NET (ADO.NET) is a model used by .NET applications, which you use to communicate with the database directly to retrieve and manipulate data.

This chapter introduces you to ADO.NET. You also learn about the concept of data binding and the different types of data binding in Windows Forms and Windows Presentation Foundation (WPF).

Let's first explore ADO.NET in brief.

Exploring ADO.NET

ADO.NET is the main data access technology that a .NET application uses. ADO.NET uses a disconnected data architecture, which means that the data you work with is just a copy of the data in the database. Microsoft chose disconnected data architecture for a number of reasons. In traditional client/server applications, you are connected to a database and you need to maintain this connection, while the application is running. However, maintaining such connections requires a lot of server resources. Therefore, when you migrate to the Internet, you should follow the disconnected data architecture, instead of maintaining direct and continuous connections with the server to reduce the load on servers.

Let's now discuss the improvements in the ADO.NET 4.0 Entity Framework and the components that constitute ADO.NET.

Improvements in the ADO.NET 4.0 Entity Framework

With ADO.NET Entity Framework, programmers are able to develop data access applications by programming the applications against a conceptual application model rather than directly programming against a relational storage schema. The main objective behind this is to decrease the amount of code and maintenance that is generally required for data-oriented applications.

The new features and enhancements incorporated in ADO.NET 4.0 Entity Framework include the following:

- **Support for creating foreign keys in the conceptual data model:** Enables developers to create conceptual models wherein foreign key columns in the database correspond to scalar properties on entity types. The properties of any entity type that map to a single field in the data storage model are referred to as scalar properties.
- **Introduction of self-tracking entities for n-tier application development:** Refers to the use of self-tracking entities while working with n-tier applications. These self-tracking entities can generally record changes to scalar, complex, and navigation properties.

Note

An n-tier application is one, which is distributed among three or more independent computers in a distributed network. In the term n-tier, n refers to some number, such as a 3-tier. In a 3-tier application, the user interface is in the user's computer, the business logic is stored in a more centralized computer, and the needed data is stored in a database server or a computer that manages a database.

- **New methods for n-tier application development:** Refers to the new methods that have been added in the System.Data.Objects namespace to facilitate easier development of n-tier applications.

- **Support for Persistence-Ignorant Objects or POCO:** Allows you to create your own custom-defined data classes that can work in conjunction with your data model. No changes to the data classes are required for this. In other words, this implies that you can use plain-old Common Language Runtime (CLR) objects (POCO), such as existing domain objects with your Entity Framework application. A POCO can be thought of as a class that does not consist of any attributes stating what responsibilities it needs to carry out. However, it does have a default constructor.
- **Lazy loading of related objects:** Implies that objects are automatically loaded from the data source when you access a navigation property. This process of lazy loading is also referred to as deferred loading.

After discussing the improvements made to ADO.NET Entity Framework, let's learn about the components of ADO.NET in the following section.

Components of ADO.NET

ADO.NET contains a set of classes that expose data access services. It provides a rich set of components to create distributed applications. ADO.NET uses a logical process flow that contains the components shown in Fig.C#-8.1:

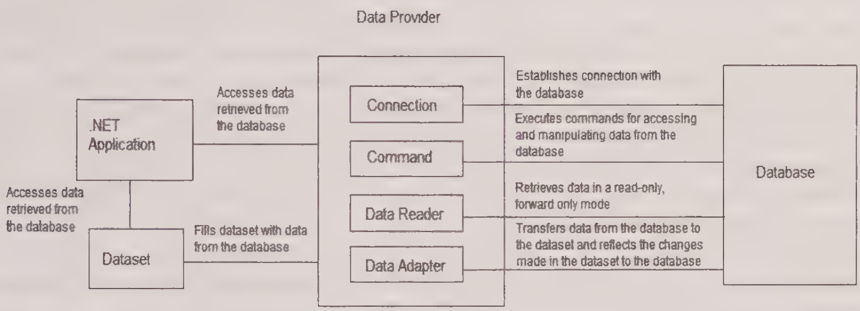


Fig.C#-8.1

As shown in Fig.C#-8.1, the two main components of ADO.NET that are used to access and manipulate data are data provider and dataset. The data provider contains the **Connection**, **Command**, **DataReader**, and **DataAdapter** objects. The **Connection** object provides connectivity to the database. The **Command** object provides access to database commands to retrieve and manipulate data in a database. The **DataReader** object retrieves data from the database in a read-only, forward-only mode. The **DataAdapter** object uses **Command** objects to execute SQL commands. The **DataAdapter** object loads the **DataSet** object with data and also resolves changes made to the data in the **DataSet** object back to the database. You learn more about data provider and the **DataSet** object in detail in the following sections.

Data Providers

A data provider is used to connect to a database, retrieve data from the database, store the data into a **DataSet** object, read the data, and finally, update the database. The data provider improves performance without compromising on functionality. Table 8.1 lists the different types of data providers included in ADO.NET:

Table 8.1: Types of ADO.NET Data Providers

Data Provider	Description
`.NET Framework Data Provider for SQL Server`	Provides access to Microsoft SQL Server 7.0 and later versions. It uses the **System.Data.SqlClient** namespace.
`.NET Framework Data Provider for Object Linking and Embedding Data Base (OLE DB)`	Provides access to databases exposed by using OLE DB. It uses the **System.Data.OleDb** namespace.
`.NET Framework Data Provider for Open Data Base Connectivity (ODBC)`	Provides access to databases exposed by using ODBC. It uses the **System.Data.Odbc** namespace.

Table 8.1: Types of ADO.NET Data Providers	
Data Provider	**Description**
`.NET Framework Data Provider for Oracle`	Provides access to Oracle database 8.1.7 and later versions. It uses the System.Data.OracleClient namespace.

The data provider contains the following four main objects:

- **Connection:** Establishes a connection with a database. The base class for all the **Connection** objects is the **DbConnection** class. **Connection** objects have methods for opening and closing a connection. The .NET Framework provides two types of Connection objects, the **SqlConnection** object, which is designed specifically to connect to Microsoft SQL Server and the **OleDbConnection** object, which is designed to provide connections to a wide range of databases, such as Microsoft Access and Oracle. Some commonly used methods of the **Connection** object are the **Open()** and **Close()** methods. The **Open()** method is used to open a connection with the database and the **Close()** method is used to close the connection.
- **Command:** Executes a command against the database and retrieves a **DataReader** or **DataSet** object. It also executes the INSERT, UPDATE, or DELETE command against the database. The base class for all **Command** objects is the **DbCommand** class. The **Command** object is represented by two classes, **SqlCommand** and **OleDbCommand**. The **Command** object provides three methods to execute commands on a database: **ExecuteNonQuery()**, **ExecuteScalar()**, and **ExecuteReader().** The **ExecuteNonQuery()** method executes commands that have no return value, such as INSERT, UPDATE, or DELETE. The **ExecuteScalar()** method returns a single value from a database query. The **ExecuteReader()** method returns a result set as a **DataReader** object.
- **DataReader:** Retrieves data from the database in a forward-only, read-only mode. The base class for all **DataReader** objects is the **DbDataReader** class. The **DataReader** object is returned when the **ExecuteReader()** method of the **Command** object is called.
- **DataAdapter:** Retrieves data from a database and stores that data in a **DataSet** object and reflects the changes made in the **DataSet** object to the database. The base class for all **DataAdapter** objects is the **DbDataAdapter** class. The **DataAdapter** object acts as an intermediary for all communication between the database and the **DataSet** object. The **DataAdapter** object is used to fill a **DataTable** or **DataSet** object with data from the database by using the **Fill()** method. The **DataAdapter** object commits the changes to the database by calling the **Update()** method. The **DataAdapter** object provides four properties that represent the database command: **SelectCommand**, **InsertCommand**, **DeleteCommand**, and **UpdateCommand**.

Datasets

The other major component of ADO.NET is the **DataSet** object. This object always remains disconnected from the database, consequently reducing the load on the database. The **DataAdapter** object is used to connect a **DataSet** object to a data provider. The **DataAdapter** object is used as an intermediary between the **DataSet** object and the data provider. The data in the **DataSet** object can be manipulated and updated independent of the database as the **DataSet** object maintains a cached copy of the data from a database.

Table 8.2 lists the various components that make up a **DataSet** object:

Table 8.2: Components of the DataSet Object	
Component	**Description**
`DataRelationCollection`	Contains a collection of DataRelation objects for a DataSet object. A DataRelation class is used to represent the parent/child relationship between tables in a database.
`ExtendedProperties`	Contains custom information, such as an SQL statement to retrieve data or the time when data was retrieved from a database
`DataTableCollection`	Contains all the tables retrieved from a database

Table 8.2: Components of the DataSet Object

Component	Description
`DataTable`	Represents a table in the **DataTableCollection** object of a **DataSet** object
`DataRelation`	Represents a relationship in the **DataRelationCollection** object of a **DataSet** object
`DataRowCollection`	Contains all the rows in a **DataTable** object
`DataView`	Represents a fixed customized view of a **DataTable** object
`PrimaryKey`	Represents a column that uniquely identifies a row in a **DataTable** object
`DataColumnCollection`	Contains all the columns in a **DataTable** object

Now, that you have learned about some of the important components of the **DataSet** object, let's discuss the basic operations that you can perform on a database by using **ADO.NET**.

Basic Operations in ADO.NET

Using ADO.NET, you can access, handle, and update data from databases in a consistent manner. In the following section, you learn to make use of the different components of ADO.NET to connect to a database, create a command, create a **DataSet** object, use a data adapter to retrieve data in a dataset, update a database by using datasets, and finally use a data reader.

Creating a Connection to a Database

To work with data from a database, you need to first establish a connection with the database. To create a new connection with a database, you can make use of the Add Connection dialog box or specify a connection string in the code-behind file. Let's first learn how to connect to a database by using the Add Connection dialog box.

Using the Add Connection Dialog Box

You can create a connection string for any data provider by using the Add Connection dialog box. In our case, we show you how to connect to the **NORTHWND** database. Let's create a connection string for Microsoft SQL Server by using the Add Connection dialog box in a Windows Forms application named **DatabaseOperationsExample**. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→ Microsoft Visual Studio 2010** to open **Microsoft Visual Studio 2010**.
2. *Select* **File→New→Project** from the menu bar. The **New Project** dialog box appears.
3. *Select* **Visual C#** under the **Installed Templates** pane and **Windows Forms Application** from the middle pane of the **New Project** dialog box.
4. *Enter* the name for the application as **DatabaseOperationsExample** and specify a location for the application in the **Location** combo box.
5. *Click* the **OK** button to create the **DatabaseOperationsExample** application.
6. *Select* **View→Server Explorer** from the menu bar to open the **Server Explorer** window, as shown in Fig.C#-8.2:

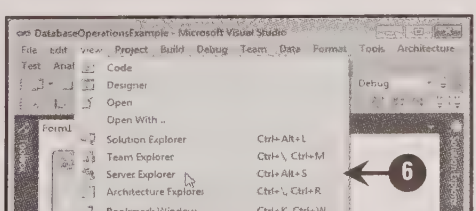


Fig.C#-8.2

The **Server Explorer** window appears (Fig.C#-8.3).

7 *Right-click* the **Data Connections** option from the **Server Explorer** window and select the **Add Connection** option from the context menu, as shown in Fig.C#-8.3:

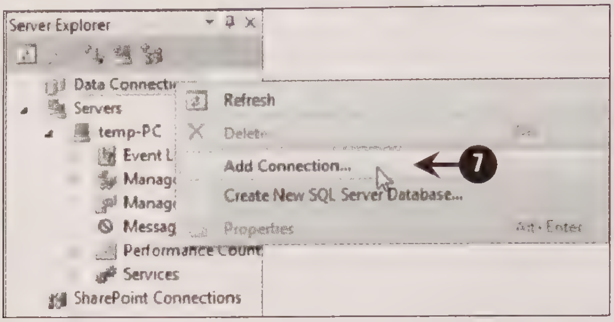


Fig.C#-8.3

The **Add Connection** dialog box appears from where you can select the type of data source you want to attach to your application.

8 *Click* the **Change** button to select the data source for your application.

The **Change Data Source** dialog box appears from where you can select the data source (Fig.C#-8.4).

9 *Select* **Microsoft SQL Server** as the data source to connect to an SQL Server database (Fig.C#-8.4).

10 *Click* the **OK** button, as shown in Fig.C#-8.4:

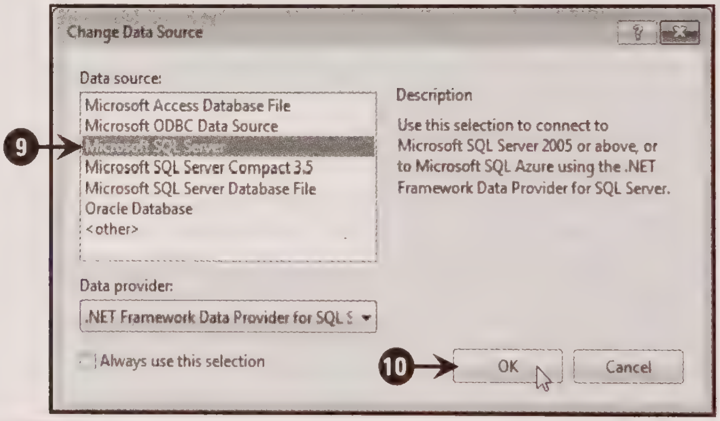


Fig.C#-8.4

The **Add Connection** dialog box reappears with the **Microsoft SQL Server (SqlClient)** option selected as a data source (Fig.C#-8.5).

11 *Select* the server name from the **Server name** drop-down list, as shown in Fig.C#-8.5:

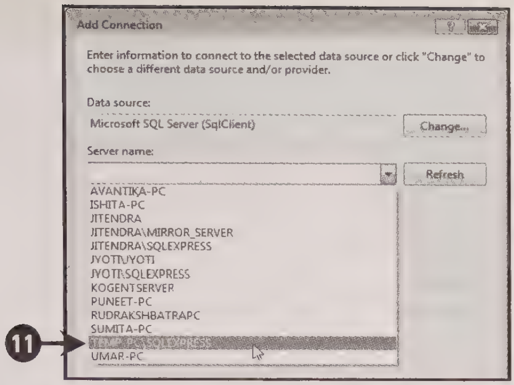


Fig.C#-8.5

By default, the **Use Windows Authentication** radio button is selected (Fig.C#-8.6). If you want to use SQL Server authentication, then select the **Use SQL Server Authentication** radio button, and provide a user name and password for the authentication. In our case, we use Windows authentication to connect to the server.

12 *Select* the database or *enter* the name of the database you want to connect to in the **Select or enter a database name** drop-down list (Fig.C#-8.6).

13 *Click* the **Test Connection** button. A message box appears with the message **Test connection succeeded** (Fig.C#-8.6).

14 *Click* the **OK** button in the message box to close the message box, as shown in Fig.C#-8.6:

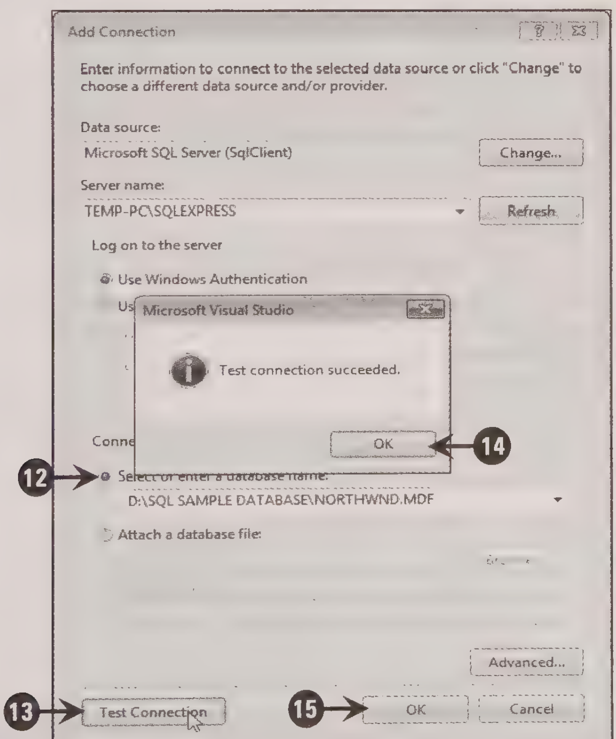


Fig.C#-8.6

15. *Click* the **OK** button of the **Add Connection** dialog box. You will notice that the data source is added to your application in **Server Explorer**, as shown in Fig.C#-8.7:

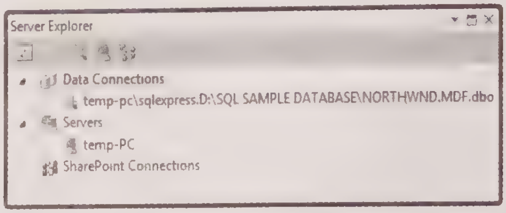


Fig.C#-8.7

Next, let's learn to establish a connection with the database by using a connection string from a code-behind file.

Using the Code-Behind File

In the preceding section, you learned to create a connection by using the Add Connection dialog box. You can also create a connection string through a code-behind file. For this, you should know the syntax of the connection string for each of the data source, which can be a database such as SQL Server, Oracle, or an Extensible Markup Language (XML) file. The connection string for an application depends on the type of data source that you select to connect with the database. The string varies according to the data source you select. For an OLE DB data source such as Microsoft Access, use the **ConnectionString** property of the **OleDbConnection** class; for SQL Server, use the **ConnectionString** property of the **SqlConnection** class; and for an Oracle data source, use the **ConnectionString** property of the **OracleConnection** class.

The basic syntax of a connection string includes a series of keywords separated by semicolons. The equal sign connects each parameter or property to its value. The following code snippet shows how to specify a connection string from the code-behind file:

```
SqlConnection con =new SqlConnection ();
con.ConnectionString = "Data Source= TEMP-PC\\SQLEXPRESS;Initial Catalog=D:\\SQL SAMPLE
DATABASE\\NORTHWND.MDF;Integrated Security=True";
```

The various parameters in the preceding code snippet can be briefly described as follows:

- **Data Source:** Represents the name or the network address of the SQL Server instance to which you need to connect.
- **Initial Catalog:** Represents the name of the database.
- **Integrated Security:** Accepts two **Boolean** values, **True** or **False**. If the value is **False,** you need to specify the user id and password in the connection string. If the value is **True,** the current Windows account credentials are used for authentication. The default value is **True**.

Until now, you have seen that you can connect to a database by using the Add Connection dialog box or by specifying the connection string in your code-behind file. However, to retrieve or manipulate data in a database you need to learn to execute a few commands by using the **Command** object. Let's learn more about this in the following section.

Executing Commands Using a Command Object

After establishing a connection with a database, you can execute commands and also return results from the database. To access a database, a data command should provide information about the connection, the SQL statement, or the name of the stored procedure to execute. The code given in Listing 8.1 shows you how to use the **Command** object in a code-behind file:

Listing 8.1: Using the Command Object

```
private void button1_Click(object sender, EventArgs e)
{
  int result;
```

```
    SqlConnection con;
    con = new SqlConnection("Data Source=TEMP-PC\\SQLEXPRESS;Initial
    Catalog= D:\\SQL SAMPLE DATABASE\\NORTHWND.MDF;Integrated Security=True");
    con.Open();
    SqlCommand command =new SqlCommand ("delete from Employees where EmployeeID=6",con);
    result=command.ExecuteNonQuery ();
}
```

In Listing 8.1, an integer variable, **result** is declared to store the result of the command executed on the data from a data source. An object of the **SqlConnection** class, **con** is also created, which takes a connection string as its argument. The connection string, **"Data Source=TEMP-PC\\SQLEXPRESS;Initial Catalog= D:\\SQL SAMPLE DATABASE\\NORTHWND.MDF;Integrated Security=True"**, opens the **NORTHWND** database on the default SQL Server instance running on the local computer. The connection to the **NORTHWND** database is established by using the **con.Open()** command. After establishing a connection with the data source, that is, the **NORTHWND** database, you need to communicate with it to execute some commands and return the result. For this, the **SqlCommand** class provides a few methods. In Listing 8.1, the **SqlCommand** class object, **command** is instantiated and accepts two parameters, namely the query to be executed in string format and the object of the **SqlConnection** class, **con**. The query specifies to delete the details of an employee whose **ID** is **6** from the **Employees** table. The **result** variable returns the result of the **ExecuteNonQuery()** method.

Let's now learn to add and configure a data adapter in the next section.

Adding and Configuring a Data Adapter

A **DataAdapter** object enables you to access data from a database by serving as a bridge between a **DataSet** object and a data source. A **DataAdapter** object establishes a connection to and retrieves data from a data source by using a **Connection** object and a **Command** object, respectively. All the commands required to manipulate, update, and modify the data in a data source are available in a **DataAdapter** object.

Let's now learn how to access data in a **DataAdapter** from a database in a Windows Forms application. In our case, we use the **DatabaseOperationsExample** application created in the **Using the Add Connection Dialog Box** section of this chapter.

Perform the following steps to access data in a **DataAdapter** object from an SQL Server database:

1. *Set* the **Text** property of **Form1** to **Basic Database Operations** in the **DatabaseOperationsExample** application.

2. *Drag* an SqlDataAdapter control from the Data tab of Toolbox and drop it into the Form1 form, if the **SqlDataAdapter** control is available in **Toolbox.**

Note

If the SqlDataAdapter control is not present in the Toolbox, perform steps 3 to 5 to add the SqlDataAdapter control to the Data tab of Toolbox; otherwise directly move to Step 6.

3. *Right-click* the **Data** tab in Toolbox and *select* the **Choose Items** option from the context menu that appears, as shown in Fig.C#-8.8:

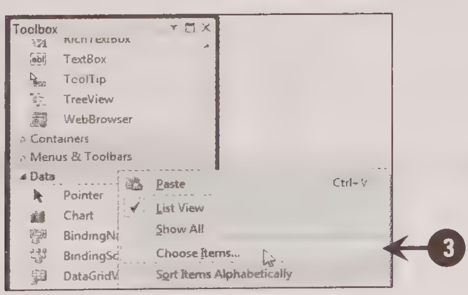


Fig.C#-8.8

The **Choose Toolbox Items** dialog box appears (Fig.C#-8.9).

4 *Select* the check box beside the **SqlDataAdapter** option under the **.NET Framework Components** tab, in the **Choose Toolbox Items** dialog box (Fig.C#-8.9).

5 *Click* the **OK** button to close the **Choose Toolbox Items** dialog box, as shown in Fig.C#-8.9:

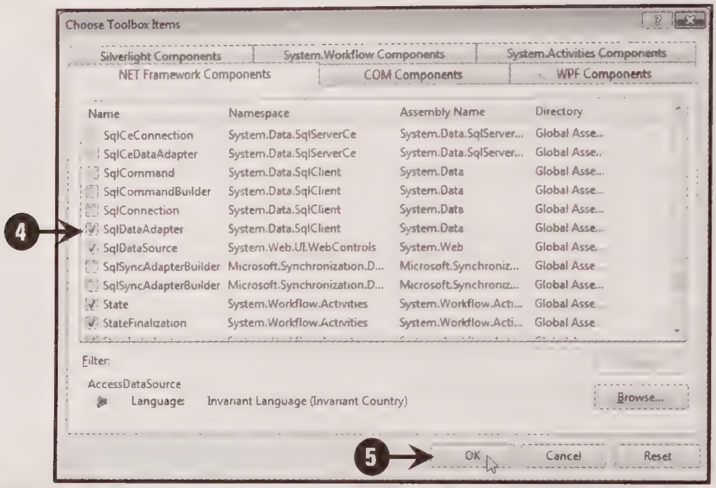


Fig.C#-8.9

The **Choose Toolbox Items** dialog box closes and the **SqlDataAdapter** control is added to **Toolbox**.

6 *Drag* and *drop* an **SqlDataAdapter** control to the **Form1** form, as shown in Fig.C#-8.10:

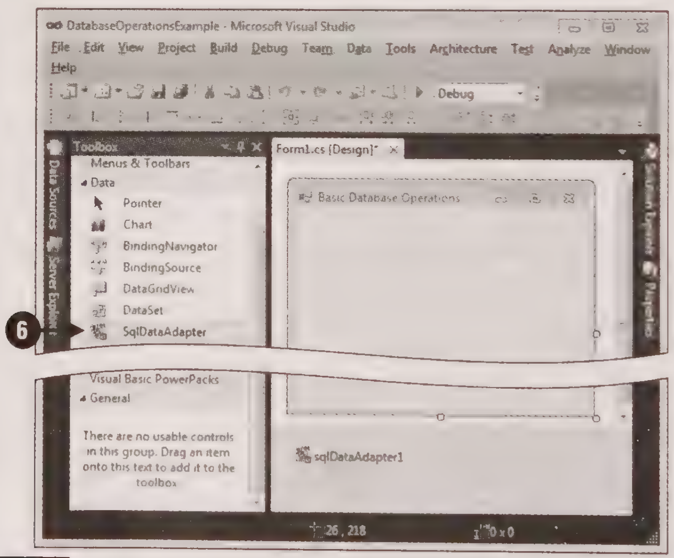


Fig.C#-8.10

As shown in Fig.C#-8.10, the **sqlDataAdapter1** data adapter is added to the component tray of the **Basic Database Operations** form.

As soon as you add the **sqlDataAdapter1** data adapter, the **Data Adapter Configuration Wizard** page appears (Fig.C#-8.11).

7 *Select* the existing connection from the **Which data connection should the data adapter use?** drop-down list or create a new connection by clicking the **New Connection** button. Clicking the **New Connection** button opens the **Add Connection** dialog box. In our case, we use the connection that we already made to the **NORTHWND** database (Fig.C#-8.11).

8 *Click* the **Next** button, as shown in Fig.C#-8.11:

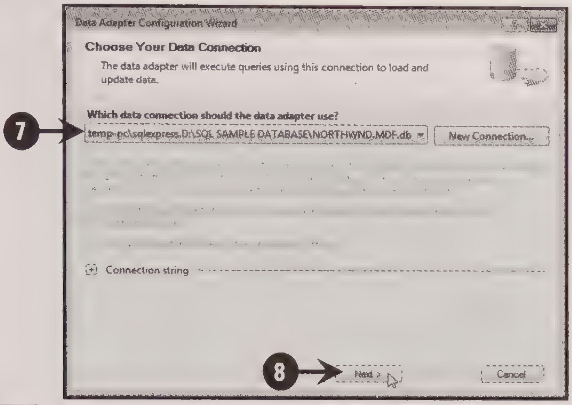


Fig.C#-8.11

The next page of Data Adapter Configuration Wizard appears, prompting you to select a command type (Fig.C#-8.12).

As you can see in Fig.C#-8.12, the Data Adapter Configuration Wizard page displays the following three radio buttons:

- **Use SQL statements:** Allows the DataAdapter object to use an SQL statement to populate a table in a dataset. This option is selected by default.
- **Create new stored procedures:** Creates a new stored procedure to read and update a table from the database. A stored procedure is a set of SQL commands compiled and stored on a database server. It can accept input parameters, perform complex actions, and return the result as output parameter to the calling procedure.
- **Use existing stored procedures:** Allows the DataAdapter object to use an existing stored procedure to read and update a table from the database.

In our case, we select the first radio button, that is, the Use SQL statements option, as shown in Fig.C#-8.12:

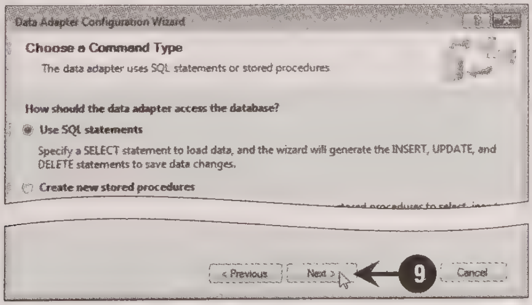


Fig.C#-8.12

9 *Click* the **Next** button (Fig.C#-8.12).

The next page of **Data Adapter Configuration Wizard** appears, prompting you to generate an SQL statement.

10. *Click* the **Query Builder** button to create an SQL statement or *select* an existing stored procedure, to generate an SQL statement.

    The Query Builder dialog box appears along with the **Add Table** dialog box (Fig.C#-8.13).

11. *Select* a table from the **Add Table** dialog box. In our case, we select the **Employees** table (Fig.C#-8.13).
12. *Click* the **Add** button to add the **Employees** table to the **Query Builder** dialog box (Fig.C#-8.13).
13. *Click* the **Close** button to close the **Add Table** dialog box, as shown in Fig.C#-8.13:

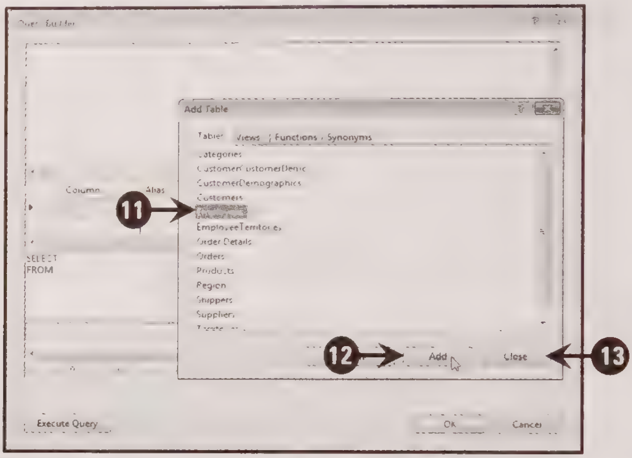


Fig.C#-8.13

The **Employees** table is added to the **Query Builder** dialog box (Fig.C#-8.14).

At the top of the **Query Builder** dialog box, you can see all the fields of the **Employees** table; you can select the fields from the table as per your choice.

14. *Select* the check box beside the ***(All Columns)** option from the **Employees** table (Fig.C#-8.14).
15. *Click* the **OK** button to create the SQL statement, as shown in Fig.C#-8.14:

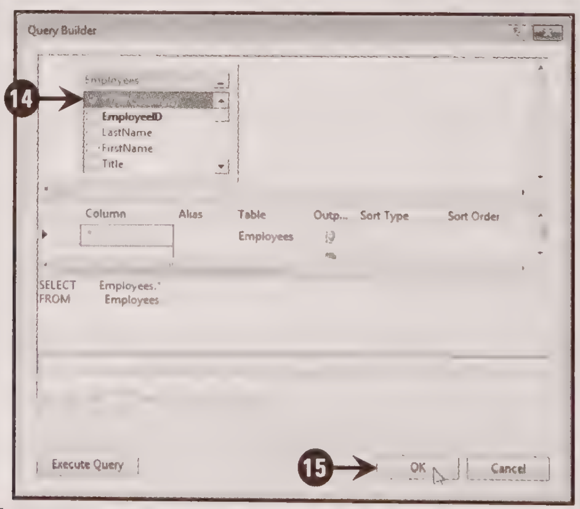


Fig.C#-8.14

The SQL statement appears in **Data Adapter Configuration Wizard** page (Fig.C#-8.15).

16 *Click* the **Next** button, as shown in Fig.C#-8.15:

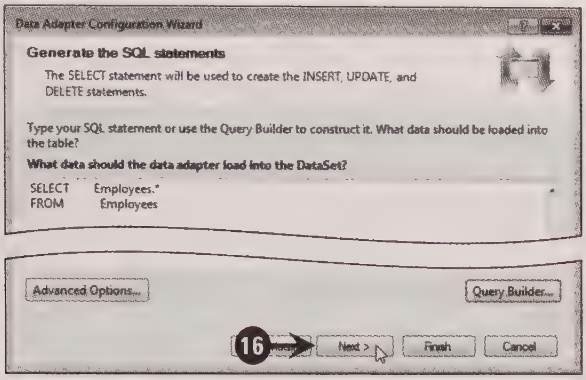


Fig.C#-8.15

The **Data Adapter Configuration Wizard** page appears with the information of a successful configuration of the data adapter.

17 *Click* the **Finish** button to close the **Data Adapter Configuration Wizard** page.

The **sqlDataAdapter1** data adapter and the **sqlConnection1** connection object appear on the component tray, as shown in Fig.C#-8.16:

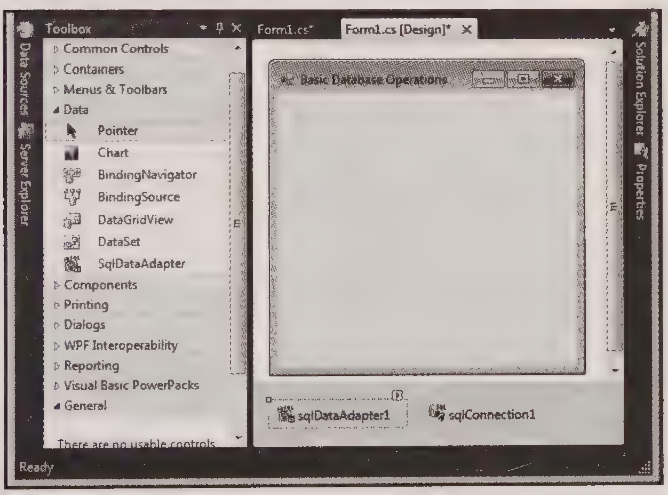


Fig.C#-8.16

18 *Select* **Data→Preview Data** from the menu bar to preview the data from the data adapter, as shown in Fig.C#-8.17:

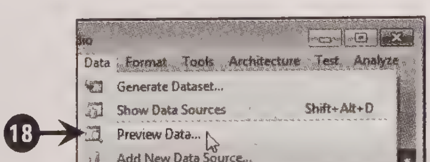


Fig.C#-8.17

The **Preview Data** dialog box appears (Fig.C#-8.18).

19 *Click* the **Preview** button to view the data from the database in the **Results** pane (Fig.C#-8.18).

20 *Click* the **Close** button to close the **Preview Data** dialog box, as shown in Fig.C#-8.18:

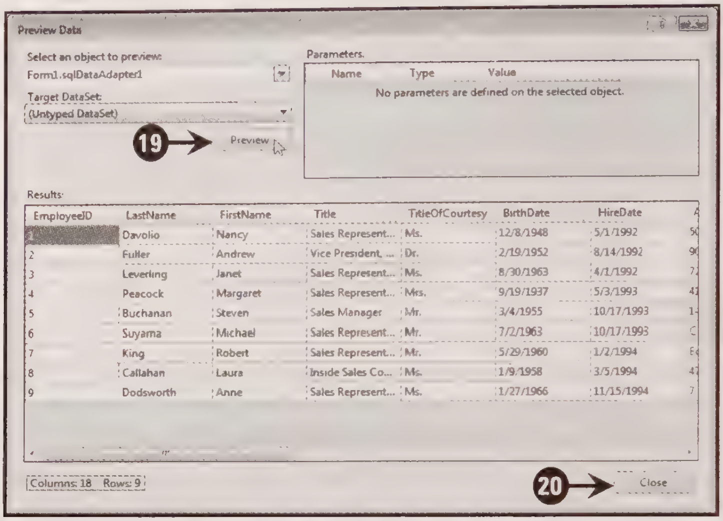


Fig.C#-8.18

Next, let's learn to display data from a data source in a Windows Forms application by using a dataset.

Creating a Dataset

Datasets, as already explained, contains a cached copy of the tables of a database. A dataset is independent of the database and so the interaction with existing databases is controlled through the data adapter. You can fill data in a dataset by using the **Fill()** method of the data adapter.

Now, let's perform the following steps to create a dataset:

1 *Open* the **DatabaseOperationsExample** application.

2 Select **Data→Generate Dataset** from the menu bar, as shown in Fig.C#-8.19:

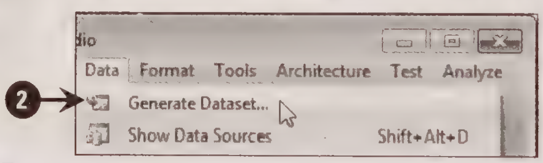


Fig.C#-8.19

The **Generate Dataset** dialog box appears, prompting you to select a dataset (Fig.C#-8.20).

You can select an existing dataset or create a new one. By default, the **New** radio button is selected. You can provide a name for the new dataset or use the existing default name as **DataSet1.** The **Generate Dataset** dialog box also prompts you to select a table to add to the dataset. In the **Generate Dataset** dialog box, you can provide a name for the dataset object and select the table(s) that you want to add to the dataset object. In our case, we have retained the default settings (Fig.C#-8.20).

3 *Click* the **OK** button to add the dataset to the **DatabaseOperationsExample** application, as shown in Fig.C#-8.20:

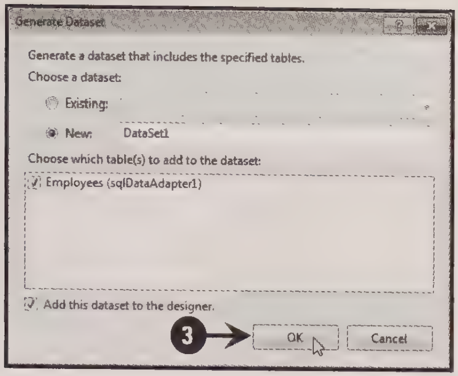


Fig.C#-8.20

The dataset, **dataSet11**, is added to the component tray, as shown in Fig.C#-8.21:

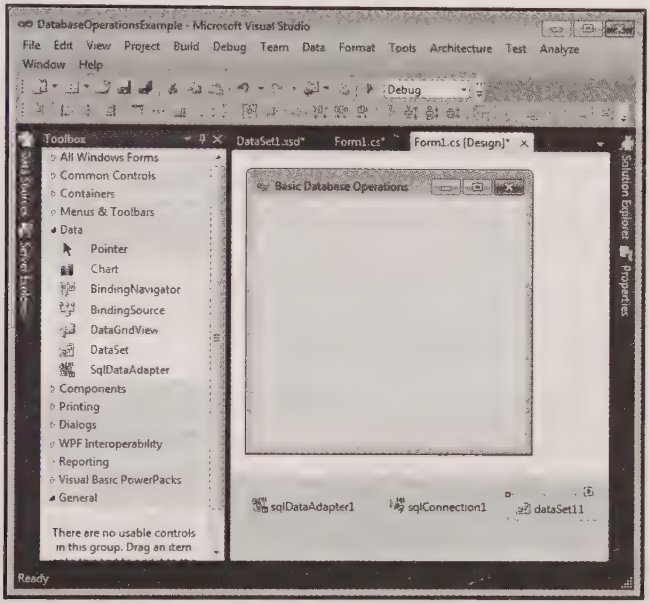


Fig.C#-8.21

After creating a dataset, let's learn how to use a data adapter to retrieve data into a dataset.

Using a Data Adapter to Retrieve Data into a Dataset

So far, we have created a data adapter and a dataset. Let's now see how to display data on a **DataGridView** control by using a data adapter and dataset. For this, perform the following steps:

1. *Open* the **DatabaseOperationsExample** application.
2. *Drag* a **DataGridView** control from the **Data** tab of **Toolbox** and *drop* it into the **Form1** form. A smart tag automatically appears along with the **DataGridView** control that you add.
3. *Click* the **Choose Data Source** drop-down list to view a list of all the available data sources (Fig.C#-8.22).
4. *Click* the triangle glyph in front of the **Other Data Sources** option to expand it (Fig.C#-8.22).
5. *Click* the triangle glyph in front of the **Project Data Sources** option to expand it (Fig.C#-8.22).
6. *Select* the **DataSet1** data source, as shown in Fig.C#-8.22:

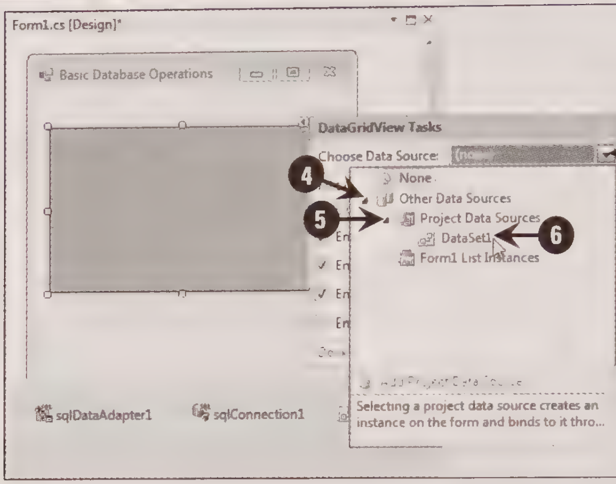


Fig.C#-8.22

The **dataSet11BindingSource** binding source is added to the component tray (Fig.C#-8.23). A binding source refers to the data source from where you can obtain data. In this case, the data is obtained by using a dataset.

7 *Click* the **Choose Data Source** down arrow and select the **Employees** table from the drop-down list that appears, as shown in Fig.C#-8.23:

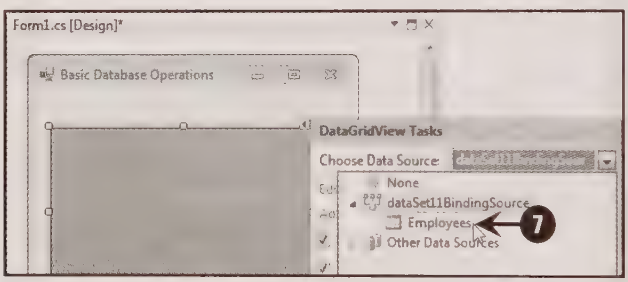


Fig.C#-8.23

The **employeesBindingSource** binding source is added to the component tray, as shown in Fig.C#-8.24:

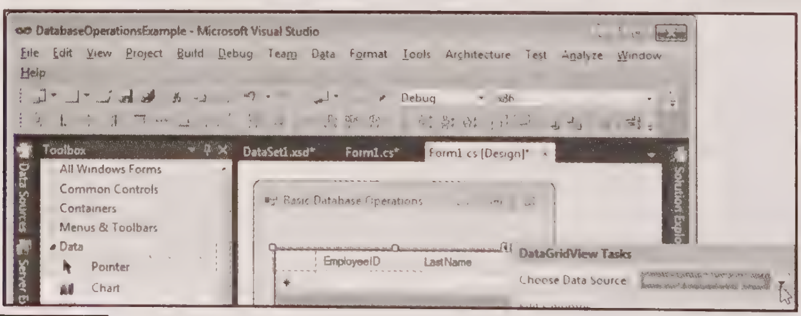


Fig.C#-8.24

8 *Resize* the form (**Basic Database Operations**) to accommodate the data of the **Employees** table.

9. *Import* the **System.Data.SqlClient** namespace into the **DatabaseOperationsExample** application by using the using statement as shown in the following code snippet:

```
using System.Data.SqlClient;
```

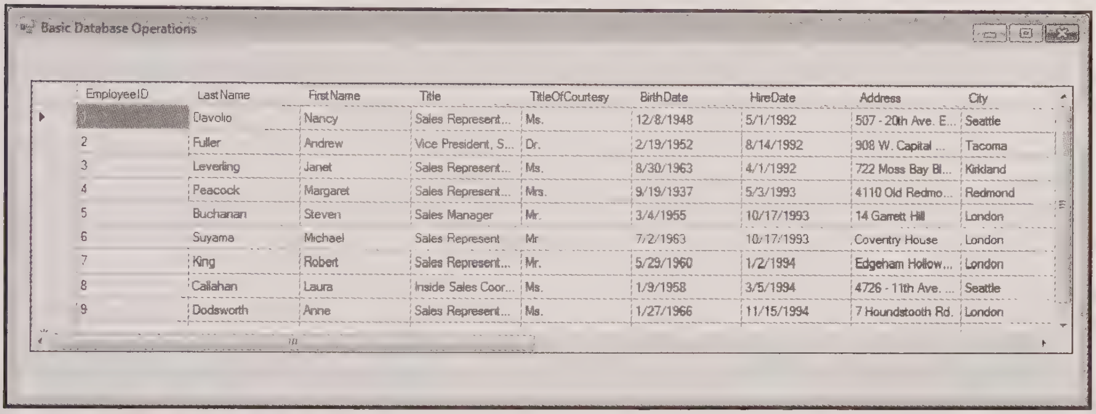
10. *Add* the highlighted code given in Listing 8.2 in the **Load** event of the **Form1** form to display employee details on a **DataGridView** control:

Listing 8.2: Populating the Dataset

```
private void Form1_Load(object sender, EventArgs e)
{
   sqlDataAdapter1.Fill(dataSet11);
}
```

Listing 8.2 shows that the **Fill()** method is used to populate the **dataSet11** dataset with employees details.

11. *Press* the **F5** key from the keyboard to execute the **DatabaseOperationsExample** application. The output appears, as shown in Fig.C#-8.25:



EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate	HireDate	Address	City
	Davolio	Nancy	Sales Represent...	Ms.	12/8/1948	5/1/1992	507 - 20th Ave. E...	Seattle
2	Fuller	Andrew	Vice President, S...	Dr.	2/19/1952	8/14/1992	908 W. Capital ...	Tacoma
3	Leverling	Janet	Sales Represent...	Ms.	8/30/1963	4/1/1992	722 Moss Bay Bl...	Kirkland
4	Peacock	Margaret	Sales Represent...	Mrs.	9/19/1937	5/3/1993	4110 Old Redmo...	Redmond
5	Buchanan	Steven	Sales Manager	Mr.	3/4/1955	10/17/1993	14 Garrett Hill	London
6	Suyama	Michael	Sales Represent	Mr	7/2/1963	10/17/1993	Coventry House	London
7	King	Robert	Sales Represent...	Mr.	5/29/1960	1/2/1994	Edgeham Hollow...	London
8	Callahan	Laura	Inside Sales Coor...	Ms.	1/9/1958	3/5/1994	4726 - 11th Ave. ...	Seattle
9	Dodsworth	Anne	Sales Represent...	Ms.	1/27/1966	11/15/1994	7 Houndstooth Rd.	London

Fig.C#-8.25

As shown in Fig.C#-8.25, the **Fill()** method of the **SqlDataAdapter** class is used to populate the **DataSet** object, **dataSet1** with data from the **NORTHWND** table of SQL Server.

Next, let's learn about the different types of data binding in Windows Forms.

Types of Data Binding in Windows Forms

Data binding refers to the binding of controls to a data from a data source to display the data. In traditional data binding, the data binding operations were mainly performed on data stored in databases. However, with Windows Forms data binding, you can now bind data from SQL Server databases and other structures, such as collections and arrays.

You can perform data binding by using a SQL Server database by binding either a particular field in a table or by binding the entire table to a control. For example, you can bind a text box to the **ProductName** field of a table or bind the entire table to a **DataGridView** control.

There are two types of data binding in Windows Forms, simple data binding and complex data binding. Let's discuss them in detail in the following sections.

Simple Data Binding Using BindingContext Class

Simple data binding allows you to display one data element, such as a field's value from a table, on a control. In Visual Basic, you can simple bind any property of a control to a data value.

For example, you can bind a **Text** property of a **TextBox** control, the **Size** and **Image** properties of a **PictureBox** control, or the **BackColor** property of a **Label** control, to a data source. You can bind a property of a control to a data source by using the **DataBindings** property of the control.

In simple terms, you can say that simple data binding is the ability of a control to bind a single data element.

Performing the following steps to illustrate simple data binding:

1. *Create* an application named **WindowsFormsDataBinding**.
2. *Set* the **Text** property of **Form1** to **Simple Data Binding**.
3. *Drag* and drop the following controls on **Form1** and set the values for their properties, as shown in Table 8.3:

Table 8.3: Controls with their Properties and Values

Control Name	Property Name	Value
label1	Text	Employee ID
label2	Text	First Name
label3	Text	Last Name
label4	Text	Title
textBox1	Enabled	False
textBox2	Enabled	False
textBox3	Enabled	False
textBox4	Enabled	False

4. *Import* the **System.Data.SqlClient** namespace in the code-behind file of the **WindowsFormsDataBinding** application by using the using statement, as shown in the following code snippet:

```
using System.Data.SqlClient;
```

5. *Add* the highlighted code given in Listing 8.3 in the **Load** event of the **Form1** form to display the details of the employee whose **EmployeeID** is **6**:

Listing 8.3: Displaying Simple Data Binding

```
private void Form1_Load(object sender, EventArgs e)
{
  SqlConnection con;
  con = new SqlConnection("Data Source=TEMP-PC\\SQLEXPRESS;Initial
  Catalog=D:\\SQL SAMPLE DATABASE\\NORTHWND.MDF;Integrated Security=True");
  con.Open();
  DataSet ds =new DataSet();
  SqlDataAdapter da = new SqlDataAdapter("Select * from Employees where EmployeeID=6 ",
  con);
  da.Fill(ds,"Employees" );
  textBox1.DataBindings.Add("Text", ds, "Employees.EmployeeID");
  textBox2.DataBindings.Add("Text", ds, "Employees.LastName");
  textBox3.DataBindings.Add("Text", ds, "Employees.FirstName");
  textBox4.DataBindings.Add("Text", ds, "Employees.Title");
}
```

In Listing 8.3, the **DataBindings** property is used to bind text boxes to the **Employees** table.

6. *Press* the **F5** key to execute the **WindowsFormsDataBinding** application. The output appears, as shown in Fig.C#-8.26:

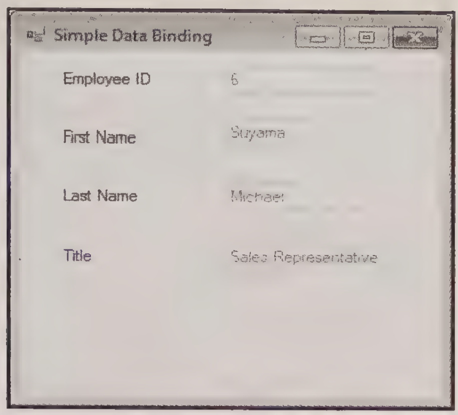


Fig.C#-8.26

In Fig.C#-8.26, the details of the employee whose **ID** is **6** are displayed in text boxes.

The **BindingContext** class in a Windows Forms application provides information about the binding and binding elements required to build the channel for data binding. You use the **BindingContext** class to access the data bindings in a control. The inheritance hierarchy of the **BindingContext** class is as follows:

```
System.Object
    System.Windows.Forms.BindingContext
```

Each object that inherits from the **Control** class can have a single **BindingContext** object. Using this object, you can access the data bindings in a form, which allows you to set the current record displayed in data-bound controls, by using the **Position** property. Let's now see an example of simple data binding by using the **BindingContext** class. For that, perform the following steps:

1. *Open* the **WindowsFormsDataBinding** application.
2. *Drag* and drop four button controls on **Form1** form and set the values for their respective properties, as shown in Table 8.4:

Table 8.4: Controls with Their Properties and Values

Control Names	Property Name	Value
button1	Text	<<
button2	Text	<
button3	Text	>
button4	Text	>>

3. *Add* the highlighted code given in Listing 8.4 in the **Form1.cs** file to display the details of all the employees:

Listing 8.4: Using the **BindingContext** Class

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.Data.SqlClient;

namespace WindowsFormsDataBinding
{
```

```
public partial class Form1 : Form
{
        SqlConnection con;
        DataSet ds = new DataSet();
        SqlDataAdapter da;
    public Form1()
        {
                InitializeComponent();
        }
        private void Form1_Load(object sender, EventArgs e)
        {
                con = new SqlConnection("Data Source=TEMP-PC\\SQLEXPRESS;Initial "+
                "Catalog=D:\\SQL SAMPLE DATABASE\\NORTHWND.MDF;
                Integrated Security=True");
                con.Open();
                da = new SqlDataAdapter("Select * from Employees", con);
                da.Fill(ds,"Employees" );
                textBox1.DataBindings.Add("Text", ds, "Employees.EmployeeID");
                textBox2.DataBindings.Add("Text", ds, "Employees.LastName");
                textBox3.DataBindings.Add("Text", ds, "Employees.FirstName");
                textBox4.DataBindings.Add("Text", ds, "Employees.Title");
        }
        private void button1_Click(object sender, EventArgs e)
        {
                this.BindingContext[ds, "Employees"].Position = 0;
        }
        private void button2_Click(object sender, EventArgs e)
        {
                this.BindingContext[ds, "Employees"].Position -= 1;
        }
        private void button3_Click(object sender, EventArgs e)
        {
                this.BindingContext[ds, "Employees"].Position += 1;
        }
        private void button4_Click(object sender, EventArgs e)
        {
                this.BindingContext[ds, "Employees"].Position = this.BindingContext[ds,
                "Employees"].Count -1;
        }
    }
}
```

In Listing 8.4, you can see that, at the **Click** event of **button1**, the **BindingContext** class has the **Position** property set to 0 to display the first record of the dataset. Similarly, the **button2** control has the **Position** property set to one less than the current position of the record, to display the previous record of the dataset. The **button3** control has a position plus one of the current position of the record, to display the next record of the dataset. Finally, the **button4** control has a position one less than the total records to display the last record of the dataset.

Note

To add the code to handle the Click events of button1, button2, button3, and button4 controls, you must double-click the button1, button2, button3, and button4 controls in the Form1 form.

4. *Press* the **F5** key to execute the **WindowsFormsDataBinding** application. The output appears.
5. *Click* the >> button, as shown in Fig.C#-8.27:

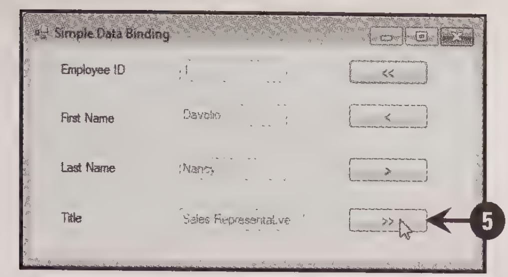


Fig.C#-8.27

The output appears, as shown in Fig.C#-8.28:

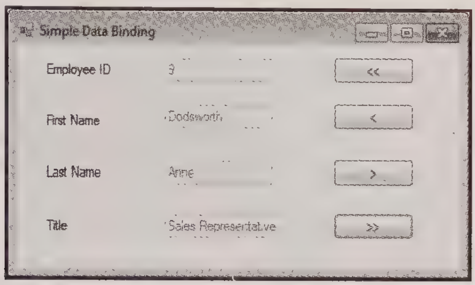


Fig.C#-8.28

As seen in Fig.C#-8.28, when you click the **>>** button, the last record of the dataset is displayed

Now, let's discuss the second type of data binding, that is, complex data binding.

Complex Data Binding

Complex data binding refers to binding of a control to more than one data element, such as more than one record in a database. Some controls that support complex data binding are **DataGridView**, **ComboBox**, **ListBox**, and **CheckedListBox**. Complex data binding can be performed by using the following properties of the controls:

- **DataSource:** Represents the data source, typically a dataset, such as **dataSet11**.
- **DataMember:** Represents the data member you want to work with, in the data source, typically a table in a dataset, such as the **Customers** table in the **NORTHWND** database. The **DataGridView** control uses this property to determine which table to display.
- **DisplayMember:** Represents the field you want a control to display, such as the customer's ID, **CustomerID**. The **ListBox** control uses the **DisplayMember** and **ValueMember** properties instead of a **DataMember** property for data binding.
- **ValueMember:** Represents the field you want a control to return in as a property, such as returning the ID of a customer by using the **SelectedValue** property. The **ListBox** control uses the **DisplayMember** and **ValueMember** properties, instead of a **DataMember** property.

Let's now illustrate how to use complex data binding by performing the following steps:

1. Open the **WindowsFormsDataBinding** application.
2. Right-click the **WindowsFormsDataBinding** application in **Solution Explorer** and *select* **Add→Windows Forms** from the context menu to add a new form to the application, as shown in Fig.C#-8.29:

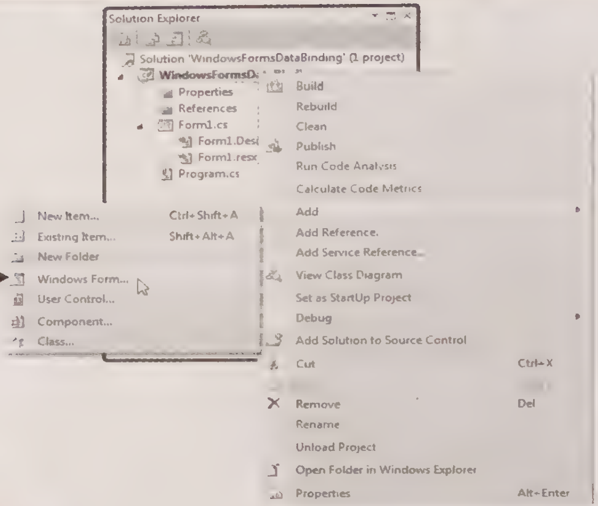


Fig.C#-8.29

The **Add New Item** dialog box appears.

3 *Click* the **Add** button to add **Form2** to the **WindowsFormsDataBinding** application.

Set the **Text** property of **Form2** to **Complex Data Binding**.

5 *Resize* the **Form2** form to accommodate all employee details.

6 *Drag* a **DataGridView** control from the **Data** tab of **Toolbox** and drop it into the **Form2** form.

7 *Import* the **System.Data.SqlClient** namespace in the **WindowsFormsDataBinding** application by using the using statement in the code-behind file, as shown in the following code snippet:

```
using System.Data.SqlClient;
```

8 *Add* the highlighted code as given in Listing 8.5 in the **Load** event of **Form2** form to display the details of all employees in a data grid view:

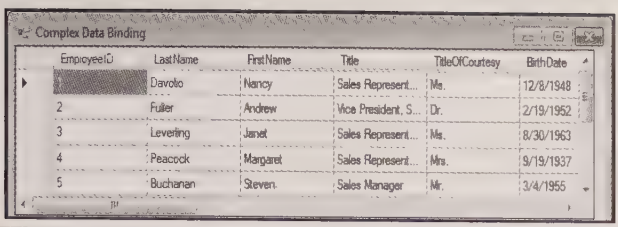
Listing 8.5: Displaying Complex Data Binding

```
private void Form2_Load(object sender, EventArgs e)
{
        SqlConnection con;
        con = new SqlConnection("Data Source=SUMITA-PC\\SQLEXPRESS;
            Initial Catalog= D:\\SQL
        SAMPLE DATABASE\\NORTHWND.MDF;Integrated Security=True");
        con.Open();
        DataSet ds = new DataSet();
        SqlDataAdapter da = new SqlDataAdapter("Select * from Employees", con);
        da.Fill(ds, "Employees");
        dataGridView1.DataSource = ds;
        dataGridView1.DataMember = "Employees";
    }
```

In Listing 8.5, the **DataSource** property is used to bind a **DataGridView** control with a data source and the **DataMember** property is used to set the table name from which the employee details need to be displayed.

9 *Set* the start up form to **Form2**.

10 *Press* the **F5** key to execute the **WindowsFormsDataBinding** application. The output appears, as shown in Fig.C#-8.30:



EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	BirthDate
	Davolio	Nancy	Sales Represent...	Ms.	12/8/1948
2	Fuller	Andrew	Vice President, S...	Dr.	2/19/1952
3	Leverling	Janet	Sales Represent...	Ms.	8/30/1963
4	Peacock	Margaret	Sales Represent...	Mrs.	9/19/1937
5	Buchanan	Steven	Sales Manager	Mr.	3/4/1955

Fig.C#-8.30

In Fig.C#-8.30, the DataGridView control, **dataGridView1** displays the details of employees from the Employees table.

You are now familiar with the data binding process in Windows Forms applications. Next, let's learn how to bind data in WPF applications.

Data Binding in WPF

As you know, WPF makes it easy to design robust and visually appealing user interfaces. You can also perform data binding in WPF applications to present and interact with data in a simple and consistent way. In WPF, you can perform data binding by using the Extensible Application Markup Language (XAML) file or the code-behind file. You can bind WPF controls and their properties to make data binding flexible and easy. For more information on WPF, refer to **Chapter 7, Introducing Windows Presentation Foundation and XAML**.

As in other applications, such as the Windows Forms and ASP.NET Web applications, WPF also needs to have a target and a source to perform data binding. You can select a public property of a control in WPF to bind your data. This may include properties of other controls, or data from a data source, such as the **NORTHWND** database. The target of the binding can be accessible to a public property of the control, for example, the **Text** property of the **TextBox** control. In short, you can say that data binding is one of the most powerful features included in WPF.

Before you learn how to perform data binding in WPF applications, let's learn about the direction of data flow in WPF applications.

Data Flow Directions

WPF data binding supports different types of binding modes between a target and a source. The data in a binding can move from the target to the source or vice versa. For example, when a user edits the contents of a **TextBox** control and its **Text** property changes, the data flows from a target to a source. Similarly, the data flows from a source to a target when the **Text** property of a **TextBox** control bound to a data source changes due to the changes made to that data source. You should specify the mode of data binding while binding data in a WPF application. The **Mode** property defines the binding mode that determines how data flows between a source and a target. There are four types of binding modes available in WPF: **OneTime** data binding, **OneWay** data binding, **TwoWay** data binding, and **OneWayToSource** data binding. Let's learn about these modes in detail in the following sections.

OneTime Data Binding

In **OneTime** data binding in WPF, the data flows from the source to the target. The binding occurs only once when the application is started or the data context changes. The best time to use **OneTime** data binding is when your data source does not implement the **INotifyPropertyChanged** interface. In other words, when you do not have to change the data or add any other data to your database, you can use **OneTime** data binding in your application. **OneTime** data binding can only retrieve data but cannot update data in the database.

OneWay Data Binding

In **OneWay** data binding in WPF, the data flows from the source to the target. This type of binding is useful for read-only data as it is not possible to change the data from the user interface. The **OneWay** binding mode in WPF is the default binding mode.

TwoWay Data Binding

The data in **TwoWay** data binding in WPF moves in both directions, that is, from the source to the target and from the target to the source. In **TwoWay** data binding, you can make changes to the data in the user interface. In this binding, the data is sent to the target, and if there is any change in the target property value, the data is sent back to the source. You can use **TwoWay** binding when you want to change the data in the user interface, which is reflected in the data source.

OneWayToSource Data Binding

In **OneWayToSource** data binding, if the target property changes, the source property is updated automatically. You can use **OneWayToSource** binding when you want to change data and get it updated in the source.

Declaration of Data Binding in WPF

You can declare binding in WPF in different ways and formats. You can create binding in the XAML format, which is the markup file of a WPF application. You can also create binding through code in the code-behind file of a WPF application. The third way to create binding in WPF is to specify the **Path** property. You can specify the source value that you want to bind by using the **Path** property. Let's now learn to perform data binding in a WPF application in these three ways.

Using XAML

You can bind a WPF application in the XAML format by specifying the **Binding** property as a markup extension. When you use the **Binding** property as an extension to declare binding, the declaration consists of a series of clauses. The clauses are in the **Name = Value** pair, where **Name** is the name of the **Binding** property and **Value** is the value you set for the property. You should note that when you create binding in the XAML format, the **Binding** property must be attached to the specific dependency property of the target object. The syntax to use the **Binding** property in the XAML format is as follows:

```
<TextBox Text="{Binding Source={StaticResource myDataSource}, Path=ContactName}"/>
```

In the preceding syntax, the **Text** property of the **TextBox** control is using the **Binding** property. Data binding in WPF provides you a simple and consistent way for the applications to present and interact with data. In WPF, you establish a binding by using the **Binding** property. Each binding in WPF has four components: binding target, binding source, target property, and a path to the source value to use.

Now, let's create a WPF browser application to perform data binding by using XAML:

1. *Create* a WPF browser application named WPFDataBinding.
2. *Set* the title of the **MainWindow** window to **Binding using XAML** by using the **Title** property.
3. *Add* the highlighted code given in Listing 8.6 to the **MainWindow.xaml** file:

Listing 8.6: Creating Data Binding by using XAML:

```
<Window x:Class=" WPFDataBinding.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title= "Binding using XAML" Height="231" Width="325">
    <Grid >
        <TextBlock Margin="10,10,10,121" FontWeight="Bold">
        Pick a color from below list
    </TextBlock>
        <ListBox Name="mcListBox" Height="100" Width="100"
              Margin="28,46,0,0" HorizontalAlignment="Left" VerticalAlignment="Top">
            <ListBoxItem>Orange</ListBoxItem>
            <ListBoxItem>Green</ListBoxItem>
            <ListBoxItem>Blue</ListBoxItem>
            <ListBoxItem>Gray</ListBoxItem>
            <ListBoxItem>LightGray</ListBoxItem>
            <ListBoxItem>Red</ListBoxItem>
```

```
        </ListBox>
        <Ellipse Height="100" Name="ellipse1" Stroke="Black" Margin="0,46,31,0"
         VerticalAlignment="Top" HorizontalAlignment="Right" Width="116">
            <Ellipse.Fill>
                <Binding ElementName="mcListBox" Path="SelectedItem.Content"/>
            </Ellipse.Fill>
        </Ellipse>
    </Grid>
</Window>
```

In Listing 8.6, the highlighted code indicates that the **Binding** property is used to bind the **Fill** property of an **Ellipse** control to the selected value of a **ListBox** control. In the code, if you see that the **Binding** property within the **Ellipse.Fill** property sets the binding from the **Ellipse** control to the **ListBox** control by specifying the control ID in the **ElementName** property and the **Path** property holds the value of the selected item in the **ListBox** control.

4. *Press* the **F5** key to execute the **WPFDataBinding** application. The output appears.
5. *Select* the **Green** color in the **ListBox** control. The output appears, as shown in Fig.C#-8.31:

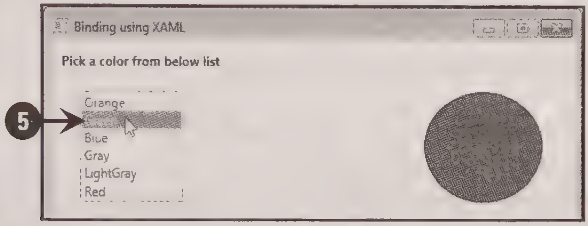


Fig.C#-8.31

In Fig.C#-8.31, when the **Green** color option is selected in the list box, the ellipse is filled with the green color.

Now, let's learn to perform data binding in WPF applications by using the code-behind file.

Using the Code-Behind File

Another way to perform data binding is to set the properties directly on the **Binding** object in code. The **FrameworkElement** and **FrameworkContentElement** classes both expose the **SetBinding()** method. You can call the **SetBinding()** method directly in your application to bind a control in code. The syntax to bind a control in a WPF application in code is as follows:

```
this.myText.SetBinding(TextBlock.TextProperty, binding1);
```

In the preceding syntax, the **Text** property of the **TextBlock** control is bound by using the **SetBinding()** method.

Now, let's create a WPF browser application to perform data binding by using the code-behind file:

1. *Open* the WPF browser application named **WPFDataBinding**.
2. *Add* a new window to the **WPFDataBinding** application by right-clicking the application name in **Solution Explorer** and selecting **Add→Window** from the context menu. The **Add New Item** dialog box appears with the name **Window1.xaml** in the **Name** text box.
3. *Click* the **OK** button. The **Window1.xaml** file is added to the **WPFDataBinding** application.
4. *Set* the title of the **Window1** window to **Binding using code-behind file** by using the **Title** property.
5. *Add* the highlighted code given in Listing 8.7 to the **Window1.xaml** file:

Listing 8.7: Adding Code for the **Window1.xaml** File

```
<Window x:Class=" WPFDataBinding.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Binding using code-behind file " Height="236" Width="300"
        Loaded="Window_Loaded">
    <Grid>
```

```
        <TextBlock Margin="10,10,10,121" FontWeight="Bold">        Pick a color from
         below list
    </TextBlock>
        <ListBox Name="mcListBox" Height="100" Width="100"         Margin="28,46,0,0"
         HorizontalAlignment="Left" VerticalAlignment="Top">
            <ListBoxItem>Orange</ListBoxItem>
            <ListBoxItem>Green</ListBoxItem>
            <ListBoxItem>Blue</ListBoxItem>
            <ListBoxItem>Gray</ListBoxItem>
            <ListBoxItem>LightGray</ListBoxItem>
            <ListBoxItem>Red</ListBoxItem>
        </ListBox>
        <Ellipse Height="100" Name="ellipse1" Stroke="Black" Margin="0,46,31,0"
         VerticalAlignment="Top" HorizontalAlignment="Right" Width="116">
        </Ellipse>
    </Grid>
</Window>
```

In Listing 8.7, we create binding through code. The **Window1.xaml** file is the same as the **MainWindow.xaml** file except for the **Binding** property used for the **Ellipse** control in the **Window1.xaml** file.

6. *Add* the highlighted code given in Listing 8.8 to the **Window1.xaml.cs** file, as shown in Listing 8.8:

Listing 8.8: Creating Data Binding by using the Code-Behind File

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
   Binding bnd = new Binding();
   bnd.Source = mcListBox;
   bnd.Path = new PropertyPath("SelectedItem.Content");
   ellipse1.SetBinding(Ellipse.FillProperty, bnd);
}
```

In Listing 8.8, a new object of the **Binding** class is created first, and then the **Source** property is used to set the binding source. Similarly, the **Path** property is used to set the property of the binding source, and finally, the **SetBinding** method is used to bind the target control to the source.

7. *Set* the **Window1** window as the startup object by double-clicking the **App.xaml** file in **Solution Explorer** and changing the **StartupUri** attribute of the **App.xaml** file to **Window1.xaml**.

8. *Press* the **F5** key to execute the **WPFDataBinding** application. The output appears.

9. *Select* the **Orange** color in the list box. The output appears, as shown in Fig.C#-8.32:

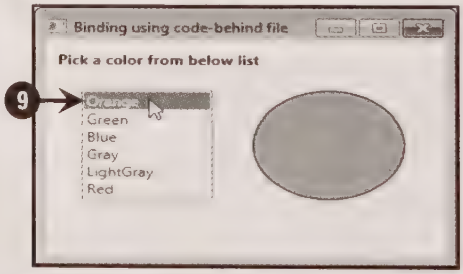


Fig.C#-8.32

In Fig.C#-8.32, when the **Orange** color option is selected from the list box, the ellipse is filled with the orange color.

Let's now learn how to perform binding by using the Path property.

Using the Path Property

You can also use the **Path** property to specify the source value that you want to bind in a WPF application. The **Path** property is the name of the property of the source object used for binding. For example, you can bind the **Text** property of the **TextBox** control. You can also bind an attached property of a control by using the **Path** property. For example, the following syntax is used with the **Path** property to bind the attached property, **DockPanel.Dock**:

```
Path = (DockPanel.Dock)
```

You can also bind the property of a control to a particular field of a database by using the **Path** property, as shown in the following syntax:

```
Path=Employees.FirstName
```

In the preceding syntax, the control is bound to the **FirstName** column of the **Employees** table.

Let's now perform data binding in a WPF application by using the **Path** property:

1. *Open* the WPF application named **WPFDataBinding**.
2. *Add* a new window to the **WPFDataBinding** application by right-clicking the application name in **Solution Explorer** and selecting **Add→Window** from the context menu. A window named **Window2.xaml** is added to the application.
3. *Set* the title of the **Window2** window to **Binding using the Path Property** by using the **Title** property.
4. *Add* the highlighted code given in Listing 8.9 to the **Window2.xaml** file:

Listing 8.9: Creating Data Binding by using the **Path** Property

```
<Window x:Class=" WPFDataBinding.Window2"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Binding using the Path Property " Height="222" Width="300" Loaded="Window_Loaded">
<Grid Name="grd" DataContext="{Binding ElementName=mcListBox, Path=SelectedItem}">
<TextBlock Margin="10,10,10,121" FontWeight="Bold">Pick a color from below list
  </TextBlock>
          <ListBox Name="mcListBox" Height="100" Width="100"
              Margin="28,46,0,0" HorizontalAlignment="Left" VerticalAlignment="Top">
              <ListBoxItem>Orange</ListBoxItem>
              <ListBoxItem>Green</ListBoxItem>
              <ListBoxItem>Blue</ListBoxItem>
              <ListBoxItem>Gray</ListBoxItem>
            <ListBoxItem>LightGray</ListBoxItem>
              <ListBoxItem>Red</ListBoxItem>
          </ListBox>
          <Ellipse Height="100" Name="ellipse1" DataContext="{Binding
              ElementName=mcListBox}" Stroke="Black" Margin="0,46,31,0"
              VerticalAlignment="Top" HorizontalAlignment="Right" Width="116">
          <Ellipse.Fill>
              <Binding Path="SelectedItem.Content"/>
          </Ellipse.Fill>
      </Ellipse>
  </Grid>
</Window>
```

In Listing 8.9, the highlighted code indicates that the **Binding** property is used to bind an **Ellipse** control's **Fill** property to a **ListBox** control's selected value. In the code, notice that the **DataContext** attribute for the **Ellipse** control is set to the binding definition. The **Binding** property of the **Ellipse** control sets the binding for the control to the **ListBox** control by specifying the ID of the control in the **ElementName** property. The value of the selected item in the **ListBox** is assigned to the **Ellipse** control by using the **Path** property.

5. *Set* **Window2** window as the startup object.
6. *Press* the **F5** key to execute the **WPFDataBinding** application. The output appears.
7. *Click* the **Blue** color option in the list box. The output appears, as shown in Fig.C#-8.33:

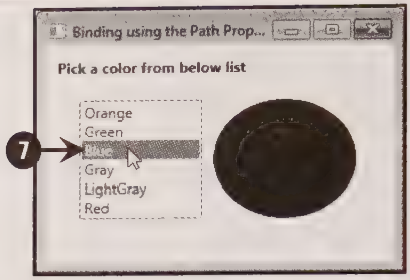


Fig.C#-8.33

In Fig.C#-8.33, the ellipse is filled with blue color, when the **Blue** option is selected in the list box.

Next, let's learn about data binding on binding sources, such as CLR and ADO.NET objects.

Binding Sources in WPF

In data binding, the source refers to the object that you obtain data from. WPF supports the CLR and ADO.NET data object binding sources. For the CLR object, data binding works as long as the binding engine is able to access the source property by using reflection. For ADO.NET object, you must establish a connection with the SQL Server database.

Note

.NET Framework facilitates you to obtain information (such as data type) about an existing variable in an application by using reflection. For instance, you can use the GetType() method to know about the data type of a variable used in an application. You can also use reflection to access the properties or invoke the methods of an existing object by using reflection in .NET applications.

Next, let's learn how to bind to CLR objects in WPF applications.

Binding to CLR Objects

In WPF, you can bind to the public properties or subproperties of any CLR object. The binding object in WPF uses CLR reflection to retrieve the values of the properties. When you use the CLR object for data binding in WPF, you should implement the **INotifyPropertyChanged** interface. This interface helps you to update the binding target when the binding source property changes. This helps to ensure that the data used in the binding stays current.

If the source object implements a proper notification mechanism, the target updates happen automatically. You can also use the **UpdateTarget()** method to update the target property to provide property change notification.

Now, let's learn how to implement CLR object binding in a WPF application. For this, perform the following steps:

1. *Create* a WPF application named **BindingtoCLRObject**.
2. *Set* the title of the **MainWindow** window to **Binding to a CLR Object** by using the **Title** property.
3. *Add* the highlighted code given in Listing 8.10 to the **MainWindow.xaml** file:

Listing 8.10: Preparing the User Interface of the **MainWindow.xaml** File

```
<Window x:Class="BindingtoCLRObject.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Binding to a CLR Object" Height="300" Width="300">
    <Grid Name="nameGrid" Margin="5,5,5,5" >
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="30*" />
            <ColumnDefinition Width="70*" />
        </Grid.ColumnDefinitions>
```

```
        <Grid.RowDefinitions>
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
            <RowDefinition Height="50" />
        </Grid.RowDefinitions>
        <Label Grid.Column="0" Grid.Row="0">Name:</Label>
        <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="0"
            Text="{Binding Name}" />
        <Label Grid.Column="0" Grid.Row="1">Age:</Label>
        <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="1"
            Text="{Binding Age}" />
        <Label Grid.Column="0" Grid.Row="2">Profile:</Label>
        <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="2"
            Text="{Binding Profile}" />
        <Label Grid.Column="0" Grid.Row="3">EmployeeID:</Label>
        <TextBox Width="200" Height="30" Grid.Column="1" Grid.Row="3"
            Text="{Binding EmployeeID}" />
        <Button Margin="5,5,5,5" Grid.Column="1" Grid.Row="4" Name="Button"
         Click="Button_Click">SHOW DATA</Button>
    </Grid>
</Window>
```

After adding the preceding code to the **MainWindow.xaml** file, you need to add a class to your application. For this, proceed to the next step.

4 *Right-click* the application name in **Solution Explorer** and *select* **Add→Class** from the context menu, as shown in Fig.C#-8.34:

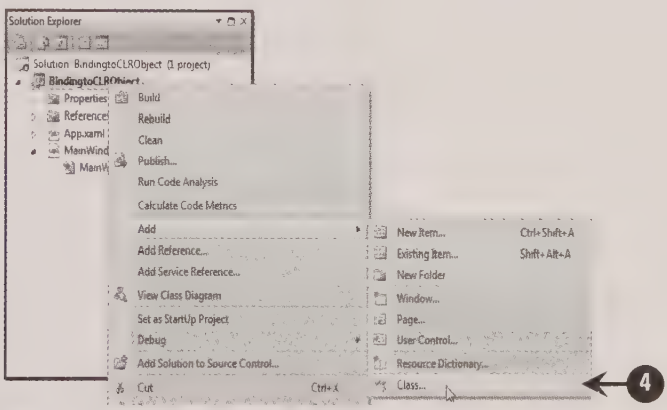


Fig.C#-8.34

The **Add New Item** dialog box appears.

5 *Enter* the name **Data.cs** in the **Name** text box.

6 *Click* the **Add** button to add the **Data.cs** file to the application.

7 *Add* the highlighted code given in Listing 8.11 to the **Data.cs** file:

Listing 8.11: Showing the Code to Add in the **Data.cs** File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace BindingtoCLRObject
{
   class Data
   {
        public Data(string name, string age, string profile, string id,
```

```
        params string[] names)
        {
                this.name = name;
                this.age = age;
                this.profile = profile;
                this.id = id;
                foreach (string Employee in names)
                {
                        this.names.Add(name);
                }
        }
        public Data()
        : this("", "", "", "")
        {
        }
        private string name;
        public string Name
        {
                get { return name; }
                set { name = value; }
        }
        private string age;
        public string Age
        {
                get { return age; }
                set { age = value; }
        }
        private string profile;
        public string Profile
        {
                get { return profile; }
                set { profile = value; }
        }
        private string id;
        public string EmployeeID
        {
                get { return id; }
                set { id = value; }
        }
        public override string ToString()
        {
                return name;
        }
        private readonly List<string> names = new List<string>();
        public string[] Names
        {
                get { return names.ToArray(); }
        }
    }
}
```

In Listing 8.11, a class named **Data** is created. The class, **Data** contains five data members, which are, **name**, **age**, **profile**, **id**, and **names**. Using properties, values have been assigned to the data members of the **Data** class.

8. *Add* the highlighted code given in Listing 8.12 to the **MainWindow.xaml.cs** file:

Listing 8.12: Showing the Code to add in the **MainWindow.xaml.cs** File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
```

```
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
namespace BindingtoCLRObject
{
  /// <summary>
  /// Interaction logic for MainWindow.xaml
  /// </summary>
  public partial class MainWindow : Window
  {
         private Data name1 = new Data();
         public MainWindow()
         {
                 InitializeComponent();
                 nameGrid.DataContext = name1;
         }
         private void Button_Click(object sender, RoutedEventArgs e)
         {
                 string message = name1.EmployeeID;
                 string caption = name1.Name;
                 MessageBox.Show(message, caption);
         }
  }
}
```

9 *Press* the **F5** key to execute the **BindingtoCLRObject** application. The output appears, as shown in Fig.C#-8.35:

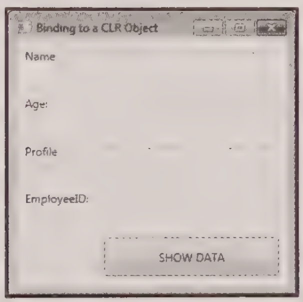


Fig.C#-8.35

In Fig.C#-8.35, you can see that the **Binding to a CLR Object** window appears in which you can enter employee details.

10 *Enter* the employee details in the text boxes (Fig.C#-8.36).

11 *Click* the **Show Data** button. A message box appears (Fig.C#-8.36).

12 *Click* the **OK** button in the message box to close it, as shown in Fig.C#-8.36:

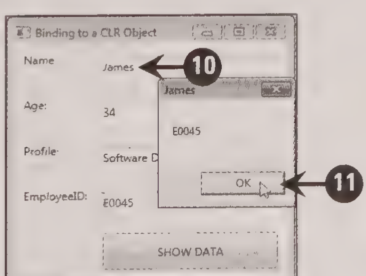


Fig.C#-8.36

As shown in Fig.C#-8.36, when you click the **SHOW DATA** button, a message box appears. The title of the message box is bound to the name of the employee you entered in the Binding to a **CLR** Object window, and it displays the ID of the employee.

In this section, you learned how to perform binding to **CLR** objects, such as properties of controls. Now, let's learn to perform binding **to ADO.NET** objects.

Binding to ADO.NET Objects

You can also bind an ADO.NET object to a WPF application. For example, you can bind a data table to a WPF application. You can implement the **IBindingList** interface to provide change notifications. The **IBindingList** interface provides features that support both complex and simple data binding to a data source. While binding your WPF application to an ADO.NET object, the first step is to create a connection with a databse. After establishing the connection, the adapter is created, which executes the SQL statement to retrieve the record from the database. The result is stored in the data table of the dataset by calling the **Fill()** method of the adapter. This result is then displayed in the WPF control.

To bind a WPF application to an ADO.NET object, you first need to create a connection string to bind your WPF application to an ADO.NET object. You then need to bind your controls by specifying the **ItemSource** and **ItemTemplate** properties. To display a particular column data, you should specify the **DisplayMemberBinding** property. Let's create an application in which you can bind an ADO.NET object to a WPF application. For this, perform the following steps:

1. *Create* a WPF application named **BindingtoADO.NET**.
2. *Set* the title of the **MainWindow** window to **Binding to ADO.NET** by using the **Title** property.
3. *Add* the highlighted code given in Listing 8.13 to the **MainWindow.xaml** file:

Listing 8.13: Adding the Code in the **MainWindow.xaml** File

```
<Window x:Class="BindingtoADO.NET.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Binding to ADO.NET" Height="300" Width="500">
       <Grid x:Name="Grid1">
       <ListView Name="ListViewEmployeeDetails" Margin="0,0,0,53"
  ItemTemplate="{DynamicResource EmployeeTemplate}" ItemsSource="{Binding
       Path=Table}"
  HorizontalAlignment="Left" Width="478" Height="172" VerticalAlignment="Bottom">
           <ListView.Background>
               <LinearGradientBrush>
                   <GradientStop Color="Bisque"  Offset="0"/>
               </LinearGradientBrush>
           </ListView.Background>
           <ListView.View>
               <GridView>
                   <GridViewColumn Header="Employee ID"
                        DisplayMemberBinding="{Binding
                 Path=EmployeeID}"/>
                   <GridViewColumn Header="First Name" DisplayMemberBinding="{Binding
                 Path=FirstName}"/>
                   <GridViewColumn Header="Last Name" DisplayMemberBinding="{Binding
                 Path=LastName}"/>
                   <GridViewColumn Header="City" DisplayMemberBinding="{Binding
                 Path=City}"/>
                   <GridViewColumn Header="Country" DisplayMemberBinding="{Binding
                 Path=Country}"/>
               </GridView>
           </ListView.View>
       </ListView>
       <Button Height="40" Margin="162,0,163,0" Name="button1"
        VerticalAlignment="Bottom"
  Click="button1_Click">Get Data</Button>
       <Label Height="28" HorizontalAlignment="Left" Margin="11,10,0,0" Name="label1"
```

```
    VerticalAlignment="Top" Width="142" FontSize="13">Data from SQL Server</Label>
    </Grid>
</Window>
```

With the help of the code given in Listing 8.13, you can prepare the interface on which you can display your data.

4. *Import* two namespaces, **System.Data** and **System.Data.SqlClient,** in the code-behind file of the **Binding to ADO.NET** application, as shown in the following code-snippet:

```
using System.Data;
using System.Data.SqlClient;
```

5. *Add* the highlighted code given in Listing 8.14 to the **MainWindow.xaml.cs** file:

Listing 8.14: Showing the Code to Add in the **MainWindow.xaml.cs** File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Data;
using System.Data.SqlClient;
namespace BindingtoADO.NET
{
   /// <summary>
   /// Interaction logic for MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window
   {
          public MainWindow()
          {
                 InitializeComponent();
          }
          private void button1_Click(object sender, RoutedEventArgs e)
          {
                 SqlConnection mycon = new SqlConnection();
                 SqlDataAdapter myadapter = new SqlDataAdapter();
                 SqlCommand cmd = new SqlCommand();
                 String dataquery = "SELECT EmployeeID, FirstName, LastName,
                        City, Country FROM Employees";
                 cmd.CommandText = dataquery;
                 myadapter.SelectCommand = cmd;
                 mycon.ConnectionString = "Data Source=TEMP-
                        PC\\SQLEXPRESS;Initial Catalog=D:\\SQL SAMPLE
                         DATABSE\\NORTHWND.MDF;
                     Integrated Security=True";
                 cmd.Connection = mycon;
                 DataSet ds = new DataSet();
                 myadapter.Fill(ds);
                 ListViewEmployeeDetails.DataContext = ds.Tables[0].DefaultView;
                 mycon.Close();
          }
   }
}
```

In Listing 8.14, we bind the WPF application to the **Employees** table of the **NORTHWND** database by using the **DataContext** object.

6. *Press* the **F5** key to execute the **BindingtoADO.NET** application. The output appears, as shown in Fig.C#-8.37:

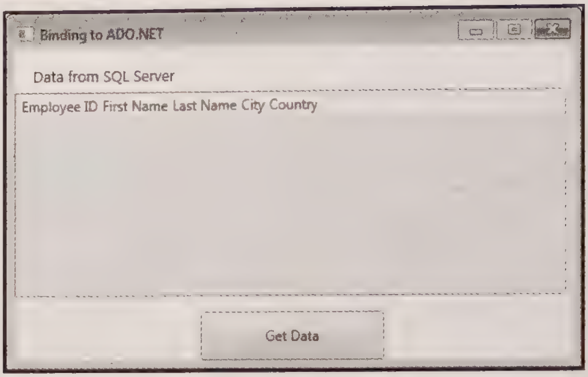


Fig.C#-8.37

7. *Click* the **Get Data** button. The output appears, as shown in Fig.C#-8.38:

Fig.C#-8.38

In Fig.C#-8.38, the employee details are displayed in a ListView control by using the DataSet object, **ds**, when the **Get Data** button is clicked.

With this, we come to the end of the chapter. Let's now recap the main topics discussed in the chapter.

Summary

In this chapter, you learned about:

- ADO.NET, its components, and the basic database operations performed by using ADO.NET
- Data binding types in Windows Forms
- Data binding in WPF applications by using the XAML or code-behind file
- Implementation of data binding in Windows Forms
- Implementation of data binding in WPF

Chapter 9

C# 2010 Delegates, Events, and Lambdas

In this Chapter:

- Working with C# Delegate Types
- Working with C# Events
- Exploring Anonymous Functions

In C and C++ function pointers are variables that directly refer to the memory location (address) of a function, which can be invoked by a function pointer. Delegates, in C# are similar to function pointers. C# delegates are object-oriented, type-safe function pointers that refer to a static or an instance method. Delegates are reference types and can be used to hold references for more than one method.

Events in C# are used to notify the user that some action has occurred, such as a mouse button has been clicked or a keyboard key has been pressed. Whenever an event is fired, an event handler to handle the event is automatically invoked. This event handling is implemented through delegates in C#.

In this chapter, you learn to work with delegates and creating single-cast and multi-cast delegates. You also learn about the event-handling mechanism in C# and the lambda expression and anonymous methods. Now, let's first start discussing how to work with delegates in the next section.

Working with C# Delegate Types

At times you may need to pass a method as a parameter to some other method. In such a situation, you can use a delegate, as it enables the developers to encapsulate a reference to a method within the delegate object. After you assign a method to a delegate, its behavior resembles that of the method. You declare a delegate in C# using the **delegate** keyword. The following is the syntax to declare a delegate:

```
delegate return_type delegate_name(parameter_list);
```

All delegate types are **sealed** and derived from the **Delegate** class. In other words you cannot define your own classes from the **Delegate** class. Following are the steps to use a delegate:

- Declare a delegate.
- Instantiate a delegate object by assigning it to a specific method.
- Invoke the delegate object by passing a single argument or a list of arguments in parenthesis.

Note

Delegates are type-safe because if you pass a method to a delegate whose signature does not match with the signature of the delegate, then, you get a compile-time error.

Now, let's learn to create a single-cast delegate in the next section.

Creating Singe-Cast Delegates

The delegates that refer to a single method are known as single-cast delegates. A single-cast delegate can be declared by using the keyword, **delegate** followed by the return type and name of the delegate. The syntax to define a single-cast delegate is as follows:

```
delegate return_type delegate_name();
```

You can prefix the delegate declaration with **public** and **internal** access modifiers. The following code snippet shows a single-cast delegate declaration using the **public** access modifier:

```
public delegate int MyDelegate(int a);
```

In the preceding code snippet, a single-cast delegate, **MyDelegate** is defined that holds reference to a method whose return type is an integer value. Any method that matches the signature of the **MyDelegate** delegate can be assigned to it. Now, whenever a method call is made to the delegate, it is automatically passed to the method it encapsulates. Moreover, the parameters that are passed to the delegate by the method caller are also passed to the method. After the method has been executed, the delegate returns a return value, if any, to the method caller. The property of a delegate to refer to a method as a parameter allows the delegate to implement callbacks

Let's now learn to create a delegate object. However, before instantiating a delegate object, you must define a method that the delegate encapsulates. Let's consider the following code snippet that consists of a method definition:

```
public int MyDelegateMethod (int x)
{
 return x * x;
}
```

After writing the preceding code snippet you need to provide the name of the method to the delegate object to instantiate it. In our case, the method name is **MyDelegateMethod()**.

The following is the code snippet to instantiate a delegate object:

```
MyDelegate delObject = MyDelegateMethod;
```

After creating the delegate object, you can call a method through the delegate and pass parameters, if any. The following code snippet demonstrates how to invoke a delegate:

```
delObject(5);
```

Now, let's create a console application named **DelegateExample** to create a single-cast delegate by performing the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from menu bar to open the **New Project** dialog box.
3. *Select* **Visual C#** in the **Installed Templates** pane and then *select* **Console Application** option in the **Templates** pane on the right-hand side.
4. *Enter* **DelegateExample** in the **Name** text box to specify the name of the application and specify the desired location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **D:\C#2010 Applications**.
5. *Click* the **OK** button. The **DelegateExample** application is created.
6. *Add* the code given in Listing 9.1 in **Program.cs** file:

Listing 9.1: Demonstrating a Single-Cast Delegate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace DelegateExample
{
 public delegate int IntDelegate(int a);
 class Calculator
 {
  public static int SquareNum(int num)
  {
   return num * num;
  }
  public static int SquareArea(int side)
  {
   return side * side;
  }
 }
 class TryDelegate
 {
  public static void Main(string[] args)
  {
   Console.WriteLine("Demonstrating a Single-Cast Delegate");
   IntDelegate delObj = new IntDelegate(Calculator.SquareNum);
   Console.WriteLine("The square of 5 is {0}", delObj(5));
   IntDelegate delObj2 = new IntDelegate(Calculator.SquareArea);
   Console.WriteLine("The area of a square with side 10 is {0}", delObj2(10));
   Console.ReadLine();
  }
 }
}
```

In Listing 9.1, we have declared a single-cast delegate called **IntDelegate**, which points to a method that accepts an integer value and returns an integer. We have defined a class, **Calculator** that contains two methods, **SquareNum()** and **SquareArea()**, whose signature matches to the signature of the delegate, **IntDelegate**. We have instantiated the single-cast delegate objects, **delObj** and **delObj2**, using the new operator and passing the method that needs to be encapsulated, as shown in the following code snippet:

```
IntDelegate delObj = new IntDelegate(Calculator.SquareNum);
IntDelegate delObj2 = new IntDelegate(Calculator.SquareArea);
```

We have invoked the single-cast delegate and passed an argument in it, as shown in the following code snippet:

```
Console.WriteLine("The square of 5 is {0}", delObj(5));
```

7. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-9.1:

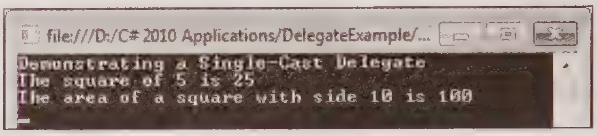


Fig.C#-9.1

In Fig.C#-9.1, you can see that the first delegate object **delObj**, returns the square of number, **5** by calling the **SquareNum()** method. Similarly, the second delegate object, **delObj2** returns the area of a square by calling the **SquareArea()** method.

Note

A callback can be thought of as a reference to a piece of executable code that is passed as an argument to some other code.

Now, let's learn about the multicast delegates next.

Creating Multi-Cast Delegates

In C#, you can create a *multicast* delegate that can hold references to more than one method. Multicast delegates derive from the **MulticastDelegate** class, which is a subclass of **Delegate** class and used to implement event handling.

The syntax to define a multicast delegate is as follows:

```
delegate void delegate_name(parameter_list);
```

To invoke a multicast delegate, the delegate object is instantiated by passing the name of the method that it encapsulates in parenthesis. The methods that the delegate object encapsulates are called in the order the methods are specified in the delegate object. A multicast delegate object maintains a list of methods, also known as an *invocation list*, which is called when the delegate is invoked. You can add method references to your delegate object using the += operator and delete method references using the -= operator. However, note that the methods that you define must not have any return type.

Now, let's create a console application named **MulticastDelegateExample** to create a multicast delegate by performing the following steps:

1. *Repeat* steps 1 to 3 of **Creating Single-Cast Delegates** section of this chapter.
2. *Enter* **MulticastDelegateExample** in the **Name** text box to specify the name of the application and specify the desired location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects.**
3. *Click* the **OK** button. The **MulticastDelegateExample** application is created.
4. *Add* the code given in Listing 9.2 in the **Program.cs** file:

Listing 9.2: Demonstrating a Multicast Delegate

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MulticastDelegateExample
{
 public class MulticastDelegateDemo
 {
  public delegate void MyMulticastDelegate();
  public static void Method1()
  {
   Console.WriteLine("Hello World");
  }
  public static void Method2()
  {
   Console.WriteLine("Demonstrating a Multicast Delegate");
  }
  public static void Method3()
  {
   Console.WriteLine("Getting Invoked by a Multicast Delegate");
  }
  public static void Main(string[] args)
  {
   MyMulticastDelegate md = new MyMulticastDelegate(Method1);
   md+=  new MyMulticastDelegate(Method2);
   md+= new MyMulticastDelegate(Method3);
   md();
   Console.ReadLine();
  }
 }
}
```

In Listing 9.2, we have declared a multicast delegate called **MyMulticastDelegate**, which can hold references to more than one target method. Note that we have defined a void delegate type, which references those methods whose return type is void.

Note

If you specify a return type for a method, which is referenced by a multicast delegate, the result of only the last method invoked by the delegate is returned. In this case, the result of all the methods with return type, void is discarded.

5. *Press* the **F5** key to run the application. The output of the **MulticastDelegateExample** application appears, as shown in Fig.C#-9.2:

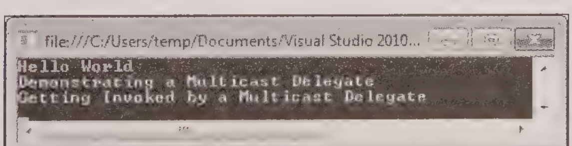


Fig.C#-9.2

In Fig.C#-9.2, note that the methods are invoked in the order they are added to the multicast delegate. You can also remove a reference from the delegate using the -= operator, as shown in the following code snippet:

```
md-=Method2;
```

On invoking the delegate object, it returns the result of **Method1** and **Method3** only, as **Method2** has been dereferenced.

Now, let's learn about events and the event handling mechanism in C# in the following section.

Working with C# Events

You have learned that delegates can be used to hold references to target methods; however, the usefulness of delegates is not restricted to only referencing methods. You can use delegates to implement event handling mechanism in C#.

In C#, an event is a mechanism through which a class is notified when an action is performed by a user or some program code or the system itself. C# is an event-driven language in which the flow of the program is controlled by user actions, such as pressing a key from the keyboard or clicking on some control with a mouse. Delegates are used to implement events in C#.

An event can be declared inside a class using the **event** keyword. The syntax to declare an event is as follows:

```
[access_modifier] event delegate_type event_identifier;
```

The class that raises or fires an event is called the **event publisher** while the class that contains the code to handle the raised event is known as **event subscriber**. Next, let's learn how events are raised and handled in C#.

Raising Events

As stated earlier, first use the **event** keyword to declare an event. Once you declare an event, you can define a delegate that is invoked when an event occurs. A delegate can refer to one or more target methods, which are called when an event is raised. Perform the following steps to create and raise events in C#:

1. Define a delegate type so that when you create a custom-defined event, there is a delegate that can be used with the event keyword.
2. Define a class that would contain an event implemented by using a delegate. The code that actually fires or raises the event is also defined in the class. Finally, a method is defined that calls the event.
3. Define one or more classes so that you can connect methods to event. In each of the classes that you define, associate the methods to the event in the class where you have defined an event using the += operator or remove a particular event using the -= operator. Also include the method definition for the methods, which you have associated with the event.
4. Finally, fire the event. Create an object of the class that contains the event declaration for firing an event. In addition, instantiate the class that includes the event definition.

Now, let's create an application named **EventHandlingExample**. Perform the following steps to raise event:

1. *Repeat* steps 1 to 3 of the **Creating Single-Cast Delegates** section of this chapter.
2. *Enter* **EventHandlingExample** in the **Name** text box to specify the name of the application and specify the desired location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects.**
3. *Click* the **OK** button. The **EventHandlingExample** application is created.
4. *Add* the code given in Listing 9.3 in **Program.cs** file:

Listing 9.3: Raising a C# Event

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace EventHandlingExample
{
  public delegate void EventDelegate();
```

```
    public class RaiseEvent
    {
     public event EventDelegate MyEvent;
     public void OnEventOccured()
     {
      if(MyEvent !=null)
      {
       MyEvent();
      }
     }
     public void FireAnEvent()
     {
      OnEventOccured();
     }
    }
  }
```

In Listing 9.3, we have defined a delegate called **EventDelegate** that refers to a method. Next, we have defined a class **RaiseEvent** that declares an event, which is implemented by the **EventDelegate** delegate. This class defines a method named **OnEventOccured()**, which raises the event. We have a method called **FireAnEvent()** that actually calls the event. Notice that in the Listing 9.3, we have checked for subscribers of the event using the following code snippet:

```
  if(MyEvent !=null)
```

If the preceding code snippet evaluates to true then the event fires.

Next, let's examine how to handle the events that are raised.

Adding Event Handlers for Raised Events

After an event has been raised, there must be a subscriber to that event. A subscriber listens to the event that has occurred and implements a method that will be invoked when the event is raised. Let's perform the following steps in the **EventHandlingExample** application to learn to handle an event that has been fired:

1. *Insert* the code given in Listing 9.4 in **Program.cs** file:

Listing 9.4: Handling a C# Event

```
  public class HandleEvent1
  {
   public void FirstEventHandler()
   {
    Console.WriteLine("Event handling inside class HandleEvent1!");
   }
  }
  class HandleEvent2
  {
   public static void SecondEventHandler()
   {
    Console.WriteLine("Event handling inside class HandleEvent2!");
   }
   public static void Main(string[] args)
   {
    RaiseEvent re = new RaiseEvent();
    HandleEvent1 he = new HandleEvent1();
    re.MyEvent+= new EventDelegate(he.FirstEventHandler);
    re.MyEvent+= new EventDelegate(SecondEventHandler);
    re.FireAnEvent();
    Console.ReadLine();
   }
  }
```

In Listing 9.4, we have defined two event handlers called **FirstEventHandler** and **SecondEventHandler** inside the classes, **HandleEvent1** and **HandleEvent2** respectively. Note that the signature of these event handlers matches to that of the delegate, **EventDelegate**. Next, we have added event handlers to the event, **MyEvent** using the += operator. Finally, we have called the method **FireAnEvent()** to fire the event.

Note

You can use the assignment operator = as well to add event handlers to an event, but in that case, all the previous references are discarded and only the last reference remains.

2. *Press* the **F5** key to run the **EventHandlingExample** application. The output appears, as shown in Fig.C#-9.3:

Fig.C#-9.3

Now, let's learn about the anonymous methods and lambda expressions in the following section.

Exploring Anonymous Functions

An *in-line* statement or expression that you use instead of a delegate type is referred to as an anonymous function. You can use an anonymous function whenever you need to initialize a named delegate or pass it instead of a named delegate type as a method parameter.

There are two kinds of anonymous functions in C#, lambda expressions and anonymous methods, which are discussed next.

Lambda Expressions

In .NET Framework 3.5, a new syntax for anonymous method was introduced with the name lambda expressions. An anonymous function that contains expressions and statements to create delegates or expression tree types is called a lambda expression. The lambda expressions uses the lambda operator **=>** (pronounced as *goes to*). The lambda operator has right-associativity and its precedence is also same as that of the assignment (=) operator. The input parameter is specified on the left side of the lambda operator and the expression or the statement is specified on the right side, as shown in the following code snippet:

```
(input parameter) => expression
```

You can omit parenthesis if the lambda expression has only one parameter. If there is more than one parameter, parenthesis is necessary. If there are more than one parameter, a comma (,) separates the parameters. If lambda expressions are used, there is no need to add a variable to the declaration as the compiler automatically adds the variable.

Note

An expression tree is basically used to represent code in a data structure similar to a tree. Each node in the tree is an expression, for instance, method invocation or binary operation such as x+y.

Using the lambda expressions, you can declare a method code inline instead of declaring a method with a delegate as lambda expressions has a concise syntax.

The lambda expressions are very helpful in writing LINQ query expressions, as they provide a very compact and safe way to write functions that are passed as arguments for further evaluation.

Now, let's create an application named **LambdaExpressionExample** by performing the following steps to learn about the lambda expression:

1. *Repeat* steps 1 to 3 of the **Creating Single-Cast Delegates** section of this chapter.
2. *Enter* **LambdaExpressionExample** in the **Name** text box to specify the name of the application and specify the desired location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects.**
3. *Click* the **OK** button. The **LambdaExpressionExample** application is created.
4. *Add* the code given in Listing 9.5 in **Program.cs** file:

Listing 9.5: Demonstrating Lambda Expression

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LambdaExpressionExample
{
    public delegate int SquareNum(int i);
    class Program
    {
       public static void Main(string[] args)
        {
             SquareNum mySquareNum = x => x * x;
             int res= mySquareNum(5);
             Console.WriteLine("The square is {0}", res);
             Console.ReadLine();
        }
    }
}
```

In Listing 9.5, you can notice that we have declared an integer delegate type called **SquareNum,** which accepts an input parameter of type int and returns an integer value. We have converted the lambda expression **x => x * x** into delegate type since it also takes a single input parameter and its return type can be implicitly converted by the compiler into int type. On invoking the delegate with input parameter **5**, it returns the result **25**.

1. *Press* the **F5** key to run the **LambdaExpressionExample** application. The output appears, as shown in Fig.C#-9.4:

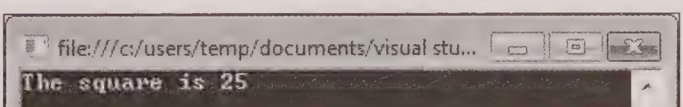


Fig.C#-9.4

You can always use a lambda expression wherever a delegate is expected.

Now, let's discuss about anonymous methods next.

Anonymous Methods

A method without any name is known as an anonymous method in C#. Anonymous methods in C# allow you to perform an action quickly. Using an anonymous method you do not need to specify a separate event handler for every event as you can specify a single anonymous method for all events. For example, if you want to display a message box at the **Click** event of a **Button** control, you can handle it in a standard way with a delegate and an event handler or you can also perform the action by using an anonymous method. In other words, an anonymous method is a block of code that is used as a parameter for a delegate. The main benefit of

using an anonymous method is that it reduces the amount of code that you need to write. When you are using an anonymous method, you do not need to define any static event handler. The anonymous method is defined at the time the caller is handling an event.

Following are the few points that should always be kept in mind while using an anonymous method:

- Do not use the **jump** statements, such as, **break**, **goto**, or **continue** inside an anonymous method having a target outside that anonymous method, as it generates an error.
- You should also not use an unsafe code inside an anonymous method.
- If you need to write the same functionality more than once, you should not use an anonymous method.

Anonymous methods were introduced in C# 2.0, which offers a new syntax for anonymous methods called lambda expression. Instead of passing anonymous methods, you can use lambda expression in Language-Integrated Query (LINQ).

Note

You learn about LINQ in Chapter 10, Introduction to Language-Integrated Query.

Summary

In this chapter, you have learned about:

- Delegates, their declaration and implementation
- Events and event handling mechanism
- Anonymous functions in C#, such as lambda expressions and anonymous methods

Chapter 10

Introduction to Language-Integrated Query

In this Chapter:

- Explaining LINQ Queries and their Execution
- Exploring Standard Query Operators
- Explaining LINQ to ADO.NET
- Exploring Parallel LINQ

Language-Integrated Query (LINQ) is a component added to .NET Framework 3.5 to provide native data-querying capabilities to the .NET Framework, by using syntax similar to that of Structured Query Language (SQL). It allows you to define statements that query a data source to yield a requested result set. LINQ provides a consistent way to obtain and manipulate data. You can use LINQ directly within C# programming language entities, called query expressions. These query expressions are based on numerous query operators that are designed to work in a manner similar to that of SQL. LINQ defines the set of query operators as the operators used to query, project, and filter data. The difference between SQL and LINQ is that unlike in SQL, query expressions in LINQ can be used to interact with numerous types of data, even with data that does not belong to a relational database. LINQ integrates the query syntax within a C# program, which makes it possible to access different data sources with the same syntax.

Apart from LINQ, this chapter introduces you to Parallel LINQ or PLINQ, which is one of the various forms in which Visual Studio 2010 and .NET Framework 4.0 offer enhanced support for parallel programming. The aim of parallel programming is to better use the ability of multi-core processors by enabling application developers to distribute their work across multiple processors.

The chapter begins by discussing LINQ queries and the standard query operators used in LINQ. You also learn about programming with LINQ to ADO.NET. Towards the end of the chapter, you learn about PLINQ, which is a new addition to .NET Framework 4.0.

Note

LINQ to ADO.NET allows you to execute queries on any enumerable data source, such as Extensible Markup Language (XML) documents, and SQL databases by using the LINQ programming model. An enumerable data source is the one that implements the IEnumerable interface.

Let's begin by learning about LINQ queries.

Explaining LINQ Queries and their Execution

A query is an expression that retrieves data from a data source. In addition, a query can be used to specify, sort, and group the retrieved data. LINQ queries are written in a specialized query language. In the past, you had to learn different languages for different data sources, for example, SQL for relational database and XQuery for Extensible Markup Language (XML). A LINQ query simplifies this situation by providing a consistent syntax to work with data across various kinds of data sources and their formats.

You can use various clauses, such as **from**, **where**, **orderby**, and **select**, with a LINQ query. These are predefined clauses that are used to execute a LINQ query.

The basic syntax of a LINQ query starts with the **from** clause and ends with the **select** or **groupby** clause. In addition, you can use the **where**, **orderby**, and **orderbydescending** clauses to perform additional functions, such as retrieving data.

The three basic steps to execute a LINQ query are as follows:

1. Obtaining the data source, which can be either a SQL database or an XML file
2. Creating a query
3. Executing the query

A simple LINQ query is executed in a **foreach** statement. This statement in C# requires the **IEnumerable** or **IEnumerable<T>** interface. A LINQ query contains three clauses: **from**, **where**, and **select**. The **from** clause specifies the data source, the **where** clause applies filter, and the **select** clause specifies the type of result.

Now, let's create a Windows application, **LINQQuery**, in which you can use a simple LINQ query to retrieve data. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar to open the **New Project** dialog box.

3. *Select* **Visual C#→Windows** in the **Installed Templates** pane and the **Windows Forms Application** option in the middle pane on the **New Project** dialog box.

4. *Enter* **LINQQuery** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.

5. *Click* the **OK** button. The **LINQQuery** application is created.

6. *Add a* **ListView** control and a **Button** control to **Form1** and change the **Text** property of **button1** to **Click**, as shown in Fig.C#-10.1:

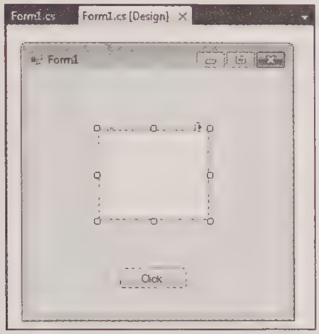


Fig.C#-10.1

7. *Add* the code given in Listing 10.1 to the **Click** event of **button1**:

Listing 10.1: Adding Code for the Click Event of the **button1** Control in the **LINQQuery** Application

```
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
  var numQuery = from num in numbers where (num % 2) == 0
  select num;
  foreach (int num in numQuery)
  {
        listView1.Items.Add(num.ToString());
  }
```

Listing 10.1 shows how the three parts of a LINQ query are expressed in a source code. The code uses an integer array as the data source.

Note

In Listing 10.1, numQuery is a LINQ query variable. A LINQ query variable itself takes no action and returns no data. It just stores the information that is required to produce the result when the query is executed.

8. *Press* the **F5** key to run the application.

9. *Click* the **Click** button, as shown in Fig.C#-10.2:

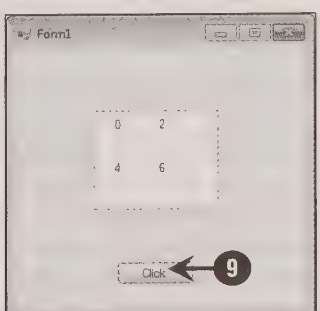


Fig.C#-10.2

In Fig.C#-10.2, you can see that on clicking the **Click** button, the **listView1** list view displays the numbers that are divisible by 2.

Now, let's discuss the standard query operators available in LINQ in the following sections.

Exploring Standard Query Operators

The standard query operators are clauses that are used to create and refine data based on a query and the needs of an application. Standard query operators can be thought of as methods that can be used to perform querying on any .NET array or collection. Some of the standard query operators include filtering, projection, partitioning, grouping, aggregation, and sorting. The standard query operators in LINQ are an Application Programming Interface (API) that enables querying of any .NET array, collection, or a sequence of data.

The various standard query operators available in LINQ differ from each other in the time they take to execute the query. The time depends on whether they have to return a single value or a sequence of values. The methods that return a single value execute immediately, while the methods that return a sequence of values reschedule the execution of the query before returning an output.

Table 10.1 lists the standard query operators used in LINQ:

Table 10.1: Standard Query Operators in LINQ

Operator Type	Description
Sorting operator	Changes the order of the elements of a sequence returned by a query
Set operator	Returns a result set in the form of a collection
Filtering operator	Restricts a result set to contain those elements that satisfy a specific condition
Quantifier operator	Returns a Boolean value, which is either True or False
Projection operator	Transforms an object into a new object
Partitioning operator	Divides the input sequence into two sections and returns one section
Join operator	Combines collections
Grouping operator	Arranges data into groups
Generation operator	Creates a new sequence of values
Element operator	Returns a single, specific element from a collection
Conversion operator	Converts a collection to an array
Aggregate operator	Computes a single value from a collection
Equality operator	Checks whether two sequences are equal or not
Concatenation operator	Appends one sequence of elements to another

Now, let's discuss the preceding standard query operators one by one in detail, in the following sections.

The Sorting Operators

The sorting operators in LINQ order the elements of a sequence based on one or more attributes. You can sort the elements of a sequence with one specific attribute and perform primary sorting on the elements. You can then specify the second sorting criterion and sort the elements within the primary sorted group. The important sorting operators in LINQ are the **orderby** and **orderbydescending** clauses.

The sorting functionality is achieved by using the **orderby** clause. A sorting operator can sort data in either ascending or descending order. The default behavior of the **orderby** clause is to sort data in ascending order. If you want to sort data in descending order, you need to use the **orderbydescending** clause.

Now, let's create a Windows application called **SortingOperator,** where you can use the **orderby** and **orderbydescending** clauses to sort data. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Enter* **SortingOperator** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **SortingOperator** application is created.
4. *Add* a **ListView** control and two **Button** controls to **Form1.**
5. *Change* the **Location** and **Size** property of the **ListView** control to **35, 31** and **214, 117**, respectively.
6. *Change* the **Text** property of the **button1** and **button2** controls to **Ascending** and **Descending**, respectively.
7. *Add* the code given in Listing 10.2 to the **Click** event of the **button1** control:

Listing 10.2: Adding Code for the **Click** Event of the **button1** Control in the **SortingOperator** Application

```
double[] doubles = { 500, 700, 100, 200, 400, 600, 900, 300, 800 };
 listView1.Items.Clear();
 var sorting = from d in doubles
        orderby d
        select d;
 foreach (var d in sorting)
 {
        listView1.Items.Add(d.ToString());
 }
```

8. *Add* the code in Listing 10.3 to the **Click** event of the **button2** control:

Listing 10.3: Adding Code for the Click Event of the **button2** Control in the **SortingOperator** Application

```
double[] doubles = { 500, 700, 100, 200, 400, 600, 900, 300, 800 };
 listView1.Items.Clear();
 var DescendingSorting = from d in doubles
        orderby d descending
        select d;
 foreach (var d in DescendingSorting)
 {
        listView1.Items.Add(d.ToString());
 }
```

The code in Listing 10.2 sorts the numbers in ascending order on clicking the **Ascending** button and the code in Listing 10.3 sorts the numbers in a descending order on clicking the **Descending** button.

9. *Press* the **F5** key to run the application.
10. *Click* the **Ascending** button, shown in Fig.C#-10.3:

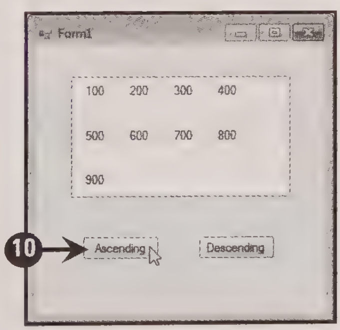


Fig.C#-10.3

As you can see in Fig.C#-10.3, the numbers are arranged in ascending order.

The Set Operators

The set operators in LINQ are used to perform query operations that yield a result set that is based on the presence or absence of equivalent elements within the same or separate collection. The **Distinct()**, **Union()**, **Intersect()**, and **Except()** methods are known as the set operators in LINQ.

The **Distinct()** method removes duplicate values from a set of values. The **Union()** method is used to return a result set, which is formed by combining the elements that appear in two sets. The result set returned by using the **Union()** method does not consist of any duplicate elements. The **Intersect()** method returns the common elements in two sets. The **Except()** method returns the elements of one set that do not appear in the second set.

The Filtering Operators

Filtering, as the name suggests, refers to the operation of filtering a result set so that it contains only those elements that satisfy a specified condition. The **where** clause is used as the filtering operator in LINQ. This clause filters a sequence based on the given condition. It is used to specify which elements from a data source are returned in a query expression.

Now, let's create a Windows application, **FilteringOperator**, where you can use the **where** clause. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the LINQQuery application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Enter* **FilteringOperator** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **FilteringOperator** application is created.
4. *Add* a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of **button1** to **Click**.
5. *Change* the **Location** and **Size** property of the **ListView** control as **12, 12** and **260, 190**, respectively.
6. *Add* the code given in Listing 10.4 to the **Click** event of the **button1** control:

Listing 10.4: Adding Code for the Click Event of the **button1** Control in the **FilteringOperator** Application

```
int[] numbers = { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70,
                   75, 80, 85, 90, 95 };
 var myNums =
 from n in numbers
 where n > 25
 select n;
 foreach (var x in myNums)
 {
        listView1.Items.Add(x.ToString());
 }
```

The code in Listing 10.4 generates the numbers that are greater than 25, which is the condition specified in the **where** clause.

7. *Press* the **F5** key to run the application.
8. *Click* the **Click** button, as shown in Fig.C#-10.4:

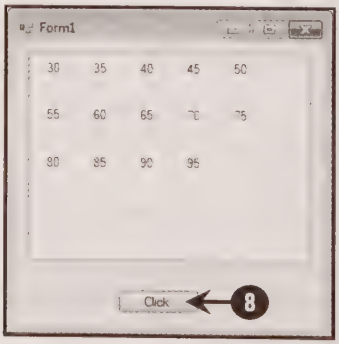


Fig.C#-10.4

As you can see in Fig.C#-10.4, the numbers that are greater than 25 appear, as per the condition specified in the **where** clause in Listing 10.4.

The Quantifier Operators

Quantifier operators return a Boolean value if the elements of a sequence satisfy a specific condition. In LINQ, the **Any()**, **All()**, and **Contains()** methods are known as quantifier operators.

The **Any()** method determines whether or not any elements in a sequence satisfy a condition. This method enumerates the source sequence and returns **True** if an element satisfies the condition. The **ArgumentNullException** exception is thrown if an argument is null.

The **All()** method determines whether or not all the elements in the sequence satisfy the given condition. This method enumerates a source sequence and returns **True** if all the elements satisfy the condition of the query. The **All()** method also returns a **True** value for an empty sequence. The **ArgumentNullException** exception is thrown if any argument is found null.

The **Contains()** method determines whether or not a sequence contains a specified element. It checks the source sequence for the element and returns a result if the element is found.

The Projection Operators

Projection operators refer to the operators used to transform an object into a new form that consists of only those properties that are subsequently used in an application. In other words, you can use projection operators to create a new object that is different from the objects used to create the object. The **select** clause and the **SelectMany()** method are the projection operators used in LINQ.

The **select** clause performs a projection over a sequence and projects the value that is based on a transform function. The **select** clause in LINQ performs the same function as that performed by the **select** statement in SQL. The **select** clause specifies which elements are to be retrieved from a data source.

The **SelectMany()** method projects sequences of values that are based on a transform function and then retrieves them into one sequence.

Now, let's create a Windows application called **ProjectionOperator,** where you can use the **select** clause. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Enter* **ProjectionOperator** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **ProjectionOperator** application is created.
4. *Add* a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of **button1** to **Select Clause.**
5. *Change* the **Location** and **Size** property of the **ListView** control to **12**, **30** and **260**, **136**, respectively.
6. *Add* the code given in Listing 10.5 to the **Click** event of the **button1** control:

Listing 10.5: Adding Code for **Click** Event of the **button1** Control in the **ProjectionOperator** Application

```
string[] text = new string[] { "Hello everybody", "this is an", "example of using",
  "select  clause", "with linq." };
      IEnumerable<string[]> words = text.Select(w => w.Split(' '));
            foreach (string[] segment in words)
                  foreach (string word in segment)
                          listView1.Items.Add(word);
```

7. *Press* the **F5** key to run the application.
8. *Click* the **Select Clause** button, as shown in Fig.C#-10.5:

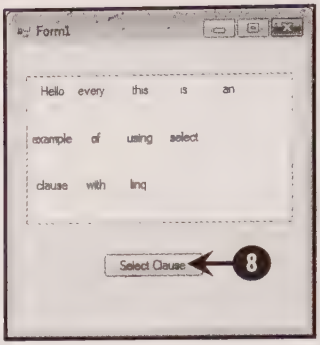


Fig.C#-10.5

In Fig.C#-10.5, each word from the string of words that were input is displayed. You can see that the words are split and displayed in different lines.

Note

Both the select clause and the SelectMany() method are used to produce a result from a data source. The difference between the two lies in the result. The select clause is used to produce one result for every value in the data source, where the result is a collection that has the same number of elements as the data source. In contrast, the SelectMany() method produces a single result that contains a concatenated collection of each element from the data source.

Now let's discuss partitioning operators in LINQ.

The Partitioning Operators

The partitioning operators in LINQ are used to divide an input sequence into two sections, without rearranging the elements, and then returning the result set with one of the sections that satisfies a given condition. The **Take()**, **Skip()**, **TakeWhile()** and **SkipWhile()** methods are known as partitioning operators in LINQ.

The **Take()** method moves elements up to a specified position in a sequence. The **TakeWhile()** method retrieves elements based on a specified position until an element satisfies the given condition. The **Skip()** method skips elements up to a specified position in a sequence. The **SkipWhile()** method skips the elements based on a given function until an element satisfies the given condition.

The Join Operators

The join operators in LINQ are used to return an output by joining the objects in one data sequence with the objects in another sequence when there is a match between the two sequences. In other words, the join operators produce a result only when common attributes are shared between two sequences. The join operators provided in LINQ are the **join** clause and **GroupJoin()** method. The **join** clause implements an inner join, which is a type of join where only those objects that have a match in other data sets are returned. The **GroupJoin()** method joins two sequences based on a **keyselector** function and groups the results.

Now, let's create a Windows application called **JoinOperator**, where you can use the **join** clause. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Enter* **JoinOperator** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **JoinOperator** application is created.
4. *Add* a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of the **button1** to **Join Data**.

5 *Change* the **Location** and **Size** property of the **ListView** control to **12**, **25** and **260**, **157**, respectively.

6 *Click* the **smart tag** of the **ListView** control and *select* the **Tile** option from the **View** drop-down list, as shown in Fig.C#-10.6:

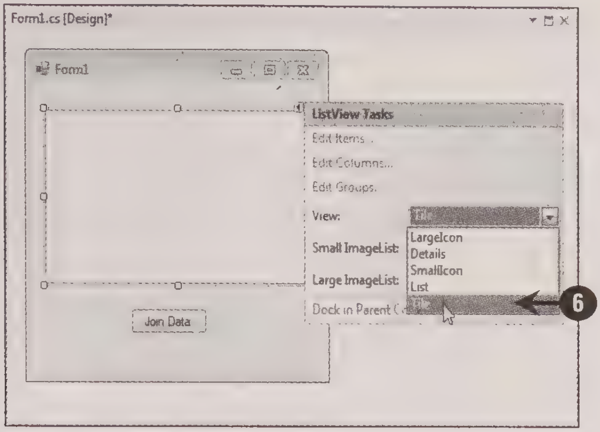


Fig.C#-10.6

7 *Add* the code given in Listing 10.6 to the **Form1.cs** file, which contains the C# code for the **Form1** form of the **JoinOperator** application:

Listing 10.6: Code to Add in the **Form1.cs** File of the **JoinOperator** Application

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace JoinOperator
{
  public partial class Form1 : Form
  {
        public Form1()
        {
                InitializeComponent();
        }
        public class Customer
        {
                public int Key;
                public string Name;
        }
        public class Order
        {
                public int Key;
                public string OrderNumber;
        }
        private void button1_Click(object sender, EventArgs e)
        {
                var customers = new List<Customer>()
                {
                        new Customer {Key = 1, Name = "Avantika" },
                        new Customer {Key = 2, Name = "Sanchita" },
                        new Customer {Key = 3, Name = "Amitabh" },
                        new Customer {Key = 4, Name = "Sumita" },
                        new Customer {Key = 5, Name = "Jitendra" }
                };
                var orders = new List<Order>()
```

```
                        {
                                new Order {Key = 1, OrderNumber = "Number 1" },
                                new Order {Key = 2, OrderNumber = "Number 2" },
                                new Order {Key = 3, OrderNumber = "Number 3" },
                                new Order {Key = 4, OrderNumber = "Number 4" },
                                new Order {Key = 5, OrderNumber = "Number 5" }
                        };
                        var q = from c in customers
                        join o in orders on c.Key equals o.Key
                        select new { c.Name, o.OrderNumber };
                        foreach (var i in q)
                        {
                                listView1.Items.Add(i.OrderNumber.ToString() + " " + i.Name);
                        }
                }
        }
    }
```

In Listing 10.6, the Customer data and the Order data are joined by using the **Join** clause.

8. *Press* the **F5** key to run the application.
9. *Click* the **Join Data** button, as shown in Fig.C#-10.7:

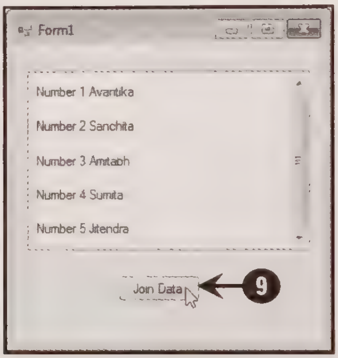


Fig.C#-10.7

Now, let's discuss the grouping operators.

The Grouping Operators

The grouping operators in LINQ are used to arrange data into groups so that the elements in each group share a common attribute. The **group** clause is the grouping operator used in LINQ. This clause returns a sequence of the **IGrouping<TKey, TElement>** objects that contain zero or more items matching the key value for the group.

Now, let's create a Windows application, **GroupingOperator**, where you can use the **group** clause. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Enter* **GroupingOperator** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **GroupingOperator** application is created.
4. *Add* a **ListView** control and a **Button** control to **Form1** and *change* the **Text** property of the **button1** to **Group Data**.
5. *Change* the **Location** and **Size** property of the **ListView** control to **12, 12** and **260, 164**, respectively.
6. *Click* the **Smart Tag** of the **ListView** control and *select* the **Tile** option from the **View** drop-down list, as shown in the **JoinOperator** example.

7. *Add* the code given in Listing 10.7 to the **Click** event of the **button1** control:

Listing 10.7: Code for the **Click** Event of the **button1** Control in the **GroupingOperator Application**

```
string[] words = { "apple", "banana", "pineapple", "apricot", "papaya", "blueberry",
  "abascus", "cherry", "parrot", "black" };
  var GroupWords = from w in words
  group w by w[0] into g
  select new { FirstLetter = g.Key, Words = g };
  foreach (var g in GroupWords)
  {
  listView1.Items.Add("Words that starts from letter: " + g.FirstLetter.ToString());
  foreach (var w in g.Words)
  {
      listView1.Items.Add(w);
  }
  }
```

8. *Press* the **F5** key to run the application.
9. *Click* the **Group Data** button, as shown in Fig.C#-10.8:

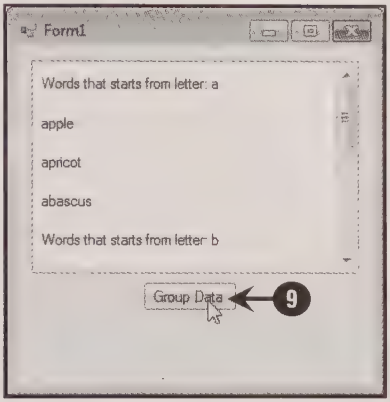


Fig.C#-10.8

In Fig.C#-10.8, you can see that the words in Listing 10.7 have been arranged alphabetically by using the Group clause.

Next, let's learn about generation operators.

The Generation Operators

Generation operators help to create a new sequence of values in a collection. The generation operators used in LINQ include the **DefaultIfEmpty()**, **Empty()**, **Range()**, and **Repeat()** methods.

The **DefaultIfEmpty()** method replaces an empty collection with the default collection. The **Empty()** method refers to an empty collection. The **Range()** method generates a collection that contains a sequence of integer numbers. This method accepts two integer type parameters, namely start and count. The start parameter refers to the value of the first number to be generated in the sequence while the count parameter specifies the total number of integer values to generate in the sequence.

Note

The Range() method throws an ArgumentOutOfRangeException exception if the value of the count parameter specified is less than 0, or if the expression start + count -1 is evaluated to a value larger than the maximum value that an integer can hold in the .NET Framework.

The **Repeat()** method generates a collection that contains at least one repeated value. The **Empty()** method caches a single empty sequence of a given type.

The Equality Operator

In LINQ, the **SequenceEqual** operator is referred to as an equality operator. This operator checks two sequences and determines whether the elements contained in them are equal or not. It returns true if all the corresponding elements in the sequences are equal and are of equal length. Otherwise, the method returns false.

Now, let's create a console application named **EqualityOperator**, where you can use the **SequenceEqual()** method. For this, perform the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from the menu bar to open the **New Project** dialog box.
3. *Select* **Visual C#** in the **Installed Templates** pane and **Console Application** in the middle pane of the **New Project** dialog box.
4. *Enter* **EqualityOperator** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box. In our case, we have selected the default location, which is **C:\Users\temp\Documents\Visual Studio 2010\Projects**.
5. *Click* the **OK** button. The **EqualityOperator** application is created.
6. *Add* the code given in Listing 10.8 inside the **Program** class of the **Program.cs** file:

Listing 10.8: Adding Code in the **Program.cs** File of the **EqualityOperator** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace EqualityOperator
{
    class EqualityOperatorDemo
    {
       public static void Main(string[] args)
        {
            var sequenceOne = new string[] {"parrot","ostrich","sparrow" };
            var sequenceTwo = new string[] {"ostrich","parrot","sparrow" };
            bool result = sequenceOne.SequenceEqual(sequenceTwo);
            Console.WriteLine("The SequenceEqual method has returned {0}", result);
            Console.ReadLine();
        }
    }
}
```

In Listing 10.8, we have used the **SequenceEqual()** method that takes two sequences, **sequenceOne** and **sequenceTwo**, and checks whether their corresponding elements match and are in the same order or not.

7. *Press* the **F5** key to run the application.

The output of the application appears, as shown in Fig.C#-10.9:

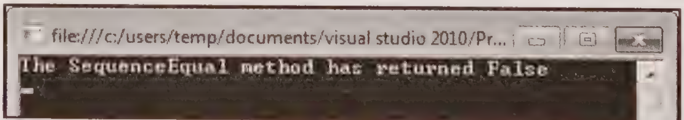


Fig.C#-10.9

In Fig.C#-10.9, you can see that the **SequenceEqual()** method has returned **False** as both the sequences passed to the method do not match.

Now, let's discuss the **Element** operators.

The Element Operators

The element operators return just one element from a data source. The element operators in LINQ include the **ElementAt()**, **ElementAtOrDefault()**, **First()**, **FirstOrDefault()**, **Last()**, **LastOrDefault()**, **Single()**, and **SingleOrDefault()** methods.

The **ElementAt()** method returns an element at a specified index in a collection. The **ElementAtOrDefault()** method returns an element at a specified index in a collection or the default value if the index is out of range. The **First()** method returns the first element of a collection or the first element that satisfies a given condition. It returns a default value if there is no such element in a collection. In addition, the **First**() method raises the **InvalidOperationException** exception when there is no element in a sequence that satisfies some specified condition. The **FirstOrDefault()** method returns the first element of a collection or the first element that satisfies a given condition. It returns a default value and unlike the **First**() method does not throw an exception if there is no such element in a collection.

The **Last()** method returns the last element of the collection or the last element that satisfies a given condition. The **LastOrDefault()** method returns the last element of a collection or the last element that satisfies a given condition. It returns a default value if there is no matching element. The **Single()** method returns the only element of a collection that satisfies a given condition. The **SingleOrDefault()** method returns the only element of a collection that satisfies a condition or the default value if there is no element in the data source.

An **ArgumentNullException** exception is thrown if any argument is null. An **InvalidOperationException** exception is thrown if no element matches a given condition or if the source sequence is empty.

The Concatenation Operator

Sometimes, you may need to merge two or more sequences into a single sequence. In such cases, you can use the **Concat** operator. The **Concat()** method is the concatenation operator in LINQ. This method concatenates two sequences and uses deferred execution.

Note

Deferred execution of a query in LINQ means the query is not executed when it is defined; instead, it is evaluated during the time it is being used.

Now, let's create a console application named **ConcatenationOperatorExample**, where you use the **Concat()** method. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **EqualityOperator** application in the **The Equality Operator** section of this chapter.
2. *Enter* **ConcatenationOperatorExample** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **ConcatenationOperatorExample** application is created.
4. *Add* the code given in Listing 10.9 in the **Program.cs** file:

Listing 10.9: Adding Code in the **Program.cs** File of the **ConcatenationOperatorExample** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ConcatenationOperatorExample
{
    class Program
    {
       public static void Main(string[] args)
        {
            string[] city = { "Jaipur","Panaji","Mumbai","Chandigarh","Shimla"};
            string[] state = { "Rajasthan","Goa","Maharashtra","Haryana",
                               "Himachal Pradesh"};
            var concatPlaces = city.Concat(state);
            Console.WriteLine("Displaying the result of Concatenation");
```

```
                foreach (var s in concatPlaces)
                {
                    Console.WriteLine(s);
                }
                Console.ReadLine();
            }
        }
    }
```

5. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-10.10:

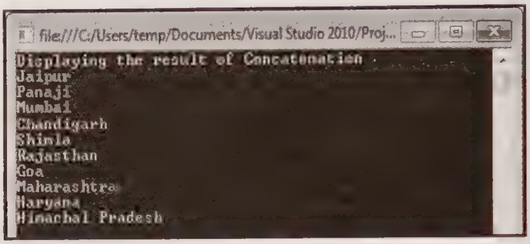


Fig.C#-10.10

In Fig.C#-10.10, you can see that the **Concat** () method merges the string arrays, **city** and **state** and displays the result as a single array of strings.

The Conversion Operators

Conversion operators convert a collection into an array and change the type of input objects. The different conversion operators used in LINQ are the **ToSequence()**, **ToArray()**, **ToList()**, **ToDictionary()**, **ToLookup()**, **OfType()**, and **Cast()** methods.

The **ToSequence()** method returns a source argument by changing it to the **IEnumerable<T>** interface. The **ToArray()** method enumerates a source sequence and returns an array containing the elements of the sequence. The **ToList()** method enumerates a source sequence and returns a List<T> interface containing the elements of the sequence. The **ToDictionary()** method lists a source sequence and evaluates the **keySelector** and **elementSelector** functions for each element to produce the key and value of the element in the source sequence. The **ToLookup()** method implements one-to-many dictionaries that map the element key to the sequence of values. The **OfType()** method allocates and returns an enumerable object that captures the source argument. The **Cast()** method is used to cast an element in a sequence to a given type. It also allocates and returns an enumerable object that captures the source argument.

The Aggregate Operators

Aggregate operators compute a single value from a collection. The different aggregate operators in LINQ are the **Aggregate()**, **Average()**, **Count()**, **LongCount()**, **Max()**, **Min()**, and **Sum()** methods. The **Aggregate()** method calculates a sum value of the values in a collection. The **Average()** method calculates the average value of the values in a collection. The **Count()** method counts the elements in a collection. The **LongCount()** method counts the elements in a large collection. The **Max()** method determines the maximum value in a collection. The **Min()** method determines the minimum value of a collection. The **Sum()** method calculates the sum of the values in a collection.

Now, let's create a console application named **AggregateOperatorExample**, to show the use of the **Sum()** method. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **EqualityOperator** application in the **The Equality Operator** section of this chapter.
2. *Enter* **AggregateOperatorExample** in the **Name** text box to specify the name of the application, and specify an appropriate location for the application in the **Location combo** box.
3. *Click* the **OK** button. The **AggregateOperatorExample** application is created.
4. *Add* the code given in Listing 10.10 in the **Program.cs** file:

Listing 10.10: Using the **Sum()** Method in the **AggregateOperatorExample** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AggregateOperatorExample
{
    class Program
    {
        public static void Main(string[] args)
        {
            int[] numArray = { 1,2,3,4,5,6,7,8,9 };
            int numSum = numArray.Sum();
            Console.WriteLine("The sum of the numbers is {0}.", numSum);
            Console.ReadLine();
        }
    }
}
```

In Listing 10.10, the **Sum()** method is used to calculate the sum of numbers.

5. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-10.11:

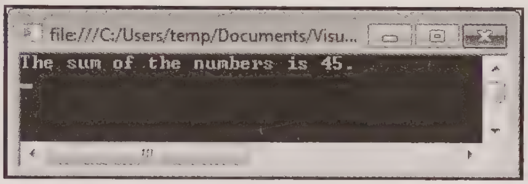


Fig.C#-10.11

Fig.C#-10.11 shows the sum of the numbers given in Listing 10.10 by using the Sum() method.

Now that you are familiar with the standard query operators available in LINQ, let's discuss programming with LINQ to ADO.NET in the next section.

Explaining LINQ to ADO.NET

LINQ to ADO.NET is a term that describes the database-centric aspects of LINQ. LINQ to ADO.NET consists of two separate technologies, LINQ to DataSet and LINQ to SQL.

LINQ to DataSet is a set of extensions to the standard ADO.NET DataSet programming model that allows **DataSet, DataTable**, and **DataRow** objects to be a natural target for a LINQ query expression. LINQ to DataSet provides optimized querying capability over datasets.

LINQ to SQL allows you to interact with a relational database by removing the ADO.NET data types through the use of entity classes. LINQ to SQL enables you to directly query SQL Server database schemas.

Now, let's discuss the LINQ to DataSet, and LINQ to SQL technologies in detail in the following sections.

Programming with LINQ to SQL

LINQ to SQL is a component that was introduced with .NET Framework 3.5. It is specifically designed to work with an SQL Server database. LINQ to SQL allows you to write queries to retrieve and manipulate data from SQL Server. LINQ to SQL supports all the key functions that you would expect while developing SQL applications. You can retrieve, insert, update, and delete data from a table of an existing SQL database. C# provides you the functionality to create LINQ to SQL classes from an existing database. It also provides with a simple way to bind Windows Forms controls to a database.

LINQ to SQL creates an Object-Relational Mapping (ORM) layer between the tables in an SQL database and the objects in a C# program. With the help of LINQ to SQL ORM mapping, the classes that match the database tables are created automatically from the database itself and you can start using the classes immediately.

The changes in LINQ to SQL in the .NET Framework 4.0 include enhancements in terms of query compilation and performance improvements. In addition to this, now you do not need to worry about encountering exceptions or facing system crashes when unexpected data types are encountered. With .NET Framework 4.0, the **LinqDataSource** control introduces support for inherited entities and types.

Note

A LinqDataSource control is a Web server control that can be used on an ASP.NET Web page. Using a LinqDataSource control helps you to connect to data on a database or in an in-memory collection of data, such as an array.

Now, let's create a Windows application, **LINQtoSQL,** to learn the implementation of LINQ to SQL. For this, perform the following steps:

1. *Repeat* steps 1 to 3 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Enter* **LINQtoSQL** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
3. *Click* the **OK** button. The **LINQtoSQL** application is created.
4. *Set* the **Size** property of **Form1** in the **Properties** window to **680, 450**, as shown in Fig.C#-10.12:

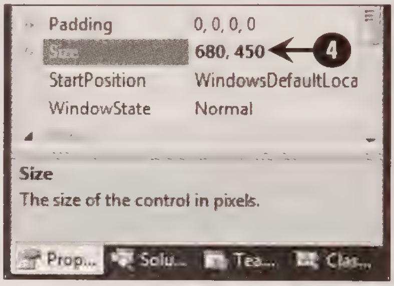


Fig.C#-10.12

5. *Add* a **ListView** control to **Form1**.
6. *Click* the **smart tag** and select the **Tile** option from the **View** drop-down list in the **ListView Tasks** menu (Fig.C#-10.13).
7. *Select* the **Dock in Parent Container** option from the **ListView Tasks** pane, as shown in Fig.C#-10.13:

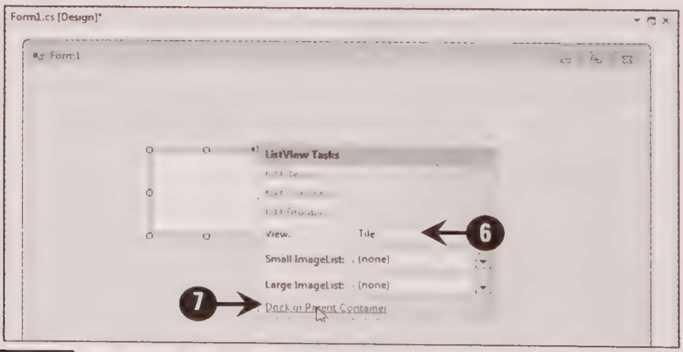


Fig.C#-10.13

After setting the properties of the **listView1** control, the next step in creating the **LINQtoSQL** application is to add a **LINQ to SQL** class. For this, let's proceed to the next step.

8 *Right-click* the project name **LINQtoSQL** in **Solution Explorer** (Fig.C#-10.14).

9 *Select* **Add→New Item** from the context menu, as shown in Fig.C#-10.14:

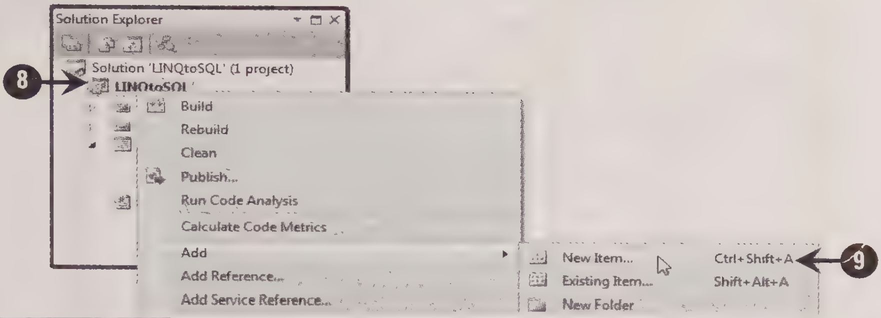


Fig.C#-10.14

The **Add New Item** dialog box opens.

10 *Select* **LINQ to SQL Classes** in the middle pane and *enter* the name **MyClass.dbml** in the **Name** text box in the **Add New Item** dialog box.

11 *Click* the **Add** button.

The **MyClass.dbml** class is added to the **LINQtoSQL** application. As you click the **Add** button, the **Object Relational Designer** window opens. You can drag tables from **Server Explorer**, drop them in the **Object Relational Designer** window, and retrieve data from the tables that you have added. In our case, we have added the **Products** table from the **NORTHWND** database.

12 *Open* the **Server Explorer** window (Fig.C#-10.15).

Tip

Alternatively, you can open the Server Explorer window by clicking the View menu on the menu bar of the Visual Studio 2010 IDE and then selecting Server Explorer from the drop-down menu that appears. You can also press the CTRL+W, L keys in combination to open the Server Explorer window.

13 *Click* the triangle glyph in front of the **Data Connections** node in the **Server Explorer** window to expand the **Data Connections** node (Fig.C#-10.15).

14 *Select* a database connection and click the triangle glyph in front of it to expand it. In our case, we have selected the **temp-pc\sqlexpress.D:\SQL SAMPLE DATABASE\NORTHWND.MDF.dbo** database connection (Fig.C#-10.15).

15 *Click* the triangle glyph in front of the **Tables** node to expand it and view the tables inside the **NORTHWND** database (Fig.C#-10.15).

16 *Drag* the **Products** table from the **Tables** node and *drop* it in the **Object Relational Designer** window, as shown in Fig.C#-10.15:

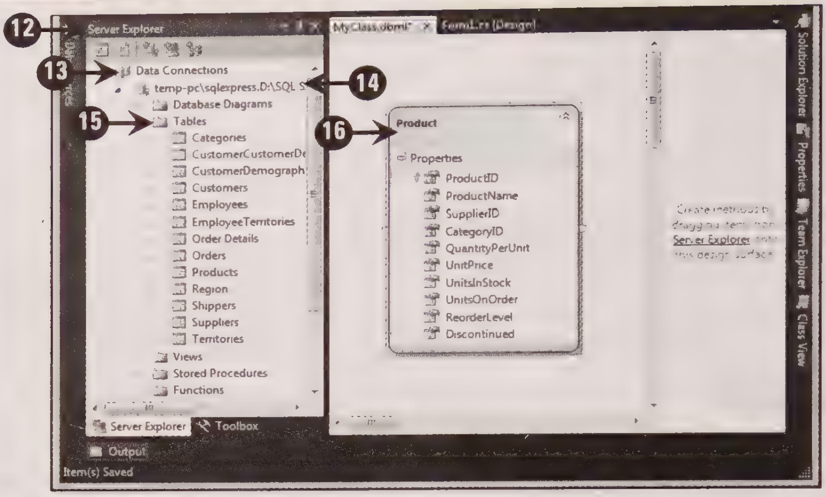


Fig.C#-10.15

17 *Select* **Build→Build Solution** from the menu bar to build the application.

18 *Add* the code given in Listing 10.11 to the **Load** event of **Form1**:

Listing 10.11: Showing Code to Add for the **Form1_Load** Event of the **LINQtoSQL** Application

```
MyClassDataContext dc = new MyClassDataContext();
  var query = dc.Products;
  foreach (Product item in query)
  {
       listView1.Items.Add(item.ProductID.ToString()+ " " +item.ProductName);
  }
```

19 *Press* the **F5** key to run the application.

The output of the **LINQtoSQL** application appears, as shown in Fig.C#-10.16:

Fig.C#-10.16

In Fig.C#-10.16, the **PoductID** and the **ProductName** properties of all the products in the **Product** table have been displayed inside the ListView control in the **Form1** form.

Now, let's learn about LINQ to DataSet programming in the next section.

Programming with LINQ to DataSet

A dataset is a very useful in-memory representation of data and acts as a core for a wide variety of data-based applications. After data is loaded into a dataset, you often need to perform additional queries on the data, and this is where LINQ to DataSet comes into play.

LINQ to DataSet enables .NET developers to query data cached in a dataset object. It simplifies querying by enabling you to write and execute queries directly from a .NET application by using any of the .NET languages, such as C# and VB.NET. You can implement LINQ to DataSet by using different query operators.

The first step before you begin querying a dataset by using LINQ to DataSet is to populate the dataset with data. After loading the data, you can begin executing queries on the data.

Now, let's create a console application called **LINQtoDataset**, to show the implementation of LINQ to DataSet, by performing the following steps:

1. *Repeat* steps 1 and 2 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and Their Execution** section of this chapter.
2. *Select* the **Visual C#→Windows** option in the **Installed Templates** pane and the **Console Application** option in the middle pane of the **New Project** dialog box.
3. *Enter* **LINQtoDataset** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
4. *Click* the **OK** button. The **LINQtoDataset** application is created.

   Now, you need to add a class to the **LINQtoDataset** application. For this, let's proceed to the next step.
5. *Right-click* the project name **(LINQtoDataset)** in **Solution Explorer** and *select* the **Add→New Item** option from the context menu. The **Add New Item** dialog box appears.
6. *Select* **Class** in the middle pane, and *enter* the name **Student.cs** in the **Name** text box of the **Add New Item** dialog box.
7. *Click* the **Add** button.
8. *Add* the code given in Listing 10.12 to the **Student.cs** file:

Listing 10.12: Showing the Code to Add to the **Student.cs** File of the **LINQtoDataset** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace LINQtoDataset
{
  class Student
  {
        public int Id;
        public string Name;
  }
}
```

9. *Add* the code given in Listing 10.13 to the **Program.cs** file of the **LINQtoDataset** application:

Listing 10.13: Adding Code to the **Program.cs** File of the **LINQtoDataset** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
namespace LINQtoDataset
{
  class Program
  {
        static DataTable GetDataTable(Student[] students)
        {
              DataTable table = new DataTable();
              table.Columns.Add("Id", typeof(Int32));
              table.Columns.Add("Name", typeof(string));
              foreach (Student s in students)
              {
```

```
                        table.Rows.Add(s.Id, s.Name);
                }
                return (table);
        }
        static void OutputDataTableHeader(DataTable dt, int ColumnWidth)
        {
                string format = string.Format("{0}0, -{1}{2}", "{", ColumnWidth, "}");
                foreach (DataColumn c in dt.Columns)
                {
                        Console.Write(format, c.ColumnName);
                }
                Console.WriteLine();
                foreach (DataColumn c in dt.Columns)
                {
                        for (int i = 0; i < ColumnWidth; i++)
                        {
                                Console.Write("=");
                        }
                }
                Console.WriteLine();
        }
        static void Main(string[] args)
        {
        Student[] students = {
                new Student {Id = 1, Name = "A"},
                new Student {Id = 2, Name = "B"},
                new Student {Id = 3, Name = "C"},
                new Student {Id = 4, Name = "D"},
                new Student {Id = 5, Name = "E"},
                new Student {Id = 6, Name = "F"}
        };
        DataTable dt = GetDataTable(students);
        Console.WriteLine("{0}Result{0}", System.Environment.NewLine);
        OutputDataTableHeader(dt, 15);
        foreach (DataRow dr in dt.Rows)
        {
                Console.WriteLine("{0,-15}{1,-15}",
                dr.Field<int>(0),
                dr.Field<string>(1));
        }
        Console.ReadLine();
        }
    }
}
```

In Listing 10.13, a set of student names is displayed in a tabular format. The tables and columns are created in the code with the help of the **DataTable** class.

10 *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-10.17:

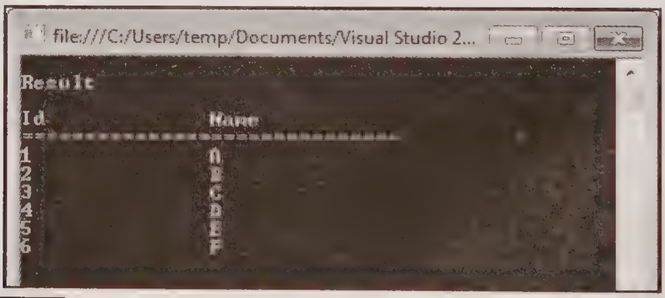


Fig.C#-10.17

In Fig.C#-10.17, you can see that the **Id** and **Name** properties of the students are displayed in a tabular format.

Let's now discuss a new feature that has been introduced in LINQ with .NET Framework 4.0. This new feature is known as Parallel LINQ or PLINQ.

Exploring Parallel LINQ

If .NET Framework 3.5 was all about LINQ, then .NET Framework 4.0 introduces the concept of PLINQ. PLINQ is nothing but a parallel implementation of the LINQ pattern and aims to use all the processors present in a system. PLINQ queries are syntactically very similar to the LINQ-to-Object queries. For instance, consider the following simple LINQ-to-Object query:

```
var output = from x in source
             where (some condition)
             select (something)
```

Now consider the following PLINQ query:

```
var output = from x in source.AsParallel()
             where (some condition)
             select(something)
```

Note that the preceding PLINQ query is the same as the LINQ-to-Object query, barring the **AsParallel()** method. This method is specific to PLINQ and is available in the **ParallelEnumerable** class. This class can be found in the **System.Linq** namespace and exposes most of the **PLINQ** functionalities.

Note

LINQ-to-Object queries refer to the execution of LINQ queries on any in-memory IEnumerable interface or IEnumerable<T> data source. In LINQ-to-Object queries, the queries are executed without using the technologies, such as LINQ to SQL.

Similar to the non-parallel sequential LINQ queries, PLINQ queries perform operations on any in-memory **IEnumerable** interface or **IEnumerable<T>** data source. PLINQ divides a data source into different segments and then executes each of these segments on separate worker threads across multiple processors. In many cases, queries run much faster because of parallel execution. However, parallelization does not imply that all the queries will run fast as there may be various complexities involved with parallelization. Moreover, note that the PLINQ concept applies only for LINQ-to-Objects queries.

Now, let's create a console application called **PLINQExample**, to execute a simple PLINQ query. For this, perform the following steps:

1. *Repeat* steps 1 and 2 shown earlier while creating the **LINQQuery** application in the **Explaining LINQ Queries and their Execution** section of this chapter.
2. *Select* the **Visual C#→Windows** option in the **Installed Templates** pane and the **Console Application** option in the middle pane of the **New Project** dialog box.
3. *Enter* **PLINQExample** in the **Name** text box to specify the name of the application, and *enter* an appropriate location for the application in the **Location** combo box.
4. *Click* the **OK** button. The **PLINQExample** application is created.
5. *Add* the code given in Listing 10.14 to the **Program.cs** file:

Listing 10.14: Adding Code to the **Program.cs** File in the **PLINQExample** Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace PLINQExample
{
    class Program
    {
       public static void Main(string[] args)
        {
            var dataSource = Enumerable.Range(2, 20);
            var output = from x in dataSource.AsParallel()
                         where x % 2 == 0
                         select x;
            foreach (var x in output)
            {
```

```
                    Console.WriteLine(x);
                }
                Console.ReadLine();
            }
        }
    }
```

Listing 10.14 shows the basic pattern to create and execute a simple PLINQ query by using the **AsParallel()** method. The **Range()** method, which is available inside the **Enumerable** class, has been used in Listing 10.14 to generate a sequence of all the integer numbers within a specified range.

6. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-10.18:

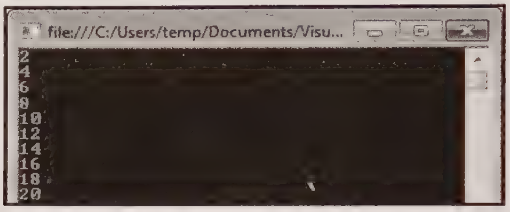


Fig.C#-10.18

In Fig.C#-10.18, we have generated a sequence of all the numbers from 2 to 20, which are divisible by 2.

With this, we come to the end of the chapter. Now, let's recapitulate all that we have learned in this chapter in a short summary.

Summary

In this chapter, you have learned about:

- Structuring of a simple LINQ query and its execution
- Executing LINQ queries by using standard query operators
- Using standard query operators in LINQ to ADO.NET
- Creating and executing a simple PLINQ query

Chapter 11

Dynamic Programming

In this Chapter:

- Describing Dynamic Language Runtime
- Exploring the Dynamic Type
- Creating the DynamicObject and ExpandoObject Class Objects
- Interoperating with Dynamic Languages

C# is a statically typed language, yet in its latest version, Microsoft has introduced the concept of dynamic programming. The reason for the support of dynamic programming being introduced in C# is due to the growth and popularity of languages, such as Ruby, Lisp, Smalltalk, and Python. C# has become more dynamic than static in nature, with the inclusion of anonymous types and methods in C# 2005 and the **var** keyword in C# 2008.

Dynamic programming is one of the most exciting and awaited features of C# 2010, which is expected to give developers more flexibility. C# 2010 implements the concept of dynamic programming with the keyword, **dynamic**. Although **dynamic** is a static type, yet if you declare any method, property, member, or operator of an object as **dynamic**, its actual type is resolved only at runtime.

In this chapter, you learn about the **Dynamic Language Runtime (DLR),** the new **dynamic** type and the instances of **DynamicObject** and **ExpandoObject** classes. You also learn about the interoperability of .NET languages with dynamic languages. Let's begin with discussing **DLR** first.

Describing Dynamic Language Runtime

As we all know that Microsoft designed the .NET Framework to support a vast array of programming languages to take advantage of the Common Language Runtime (CLR). The CLR provides a host of services that can be implemented by any of the .NET languages. Today, CLR supports dynamic languages because of the addition of DLR on top of CLR. DLR contains a small set of services that are explicitly designed to cater the needs of dynamic languages. The services provided by DLR include shared dynamic type system, standard hosting model, and support for faster generation of dynamic code. These services can be implemented by all dynamic languages that use DLR to share their code with some other dynamic languages such as Ruby and Python as well as with existing .NET languages such as VB.NET and C#.

DLR is added on top of CLR so that the creation and utilization of dynamic languages in .NET Framework can be simplified and the features of dynamic languages can be used in C# as well. You can find DLR within the .NET Framework in the System.Dynamic namespace. Fig.C#-11.1 shows the architecture and features of DLR:

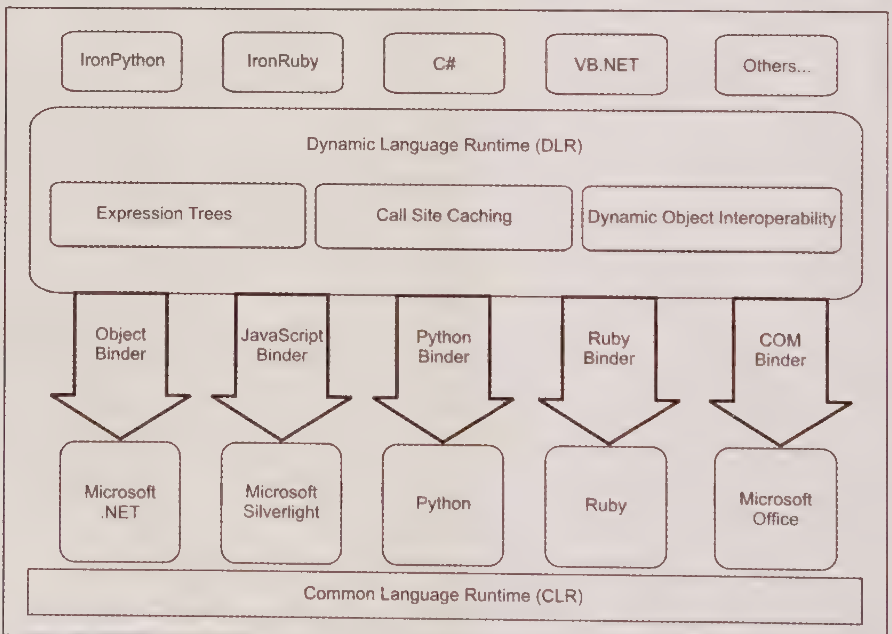


Fig.C#-11.1

In Fig.C#-11.1, there are several languages at the top, such as IronPython, IronRuby, C#, VB.NET, and others in which you can write the source code. At the bottom, there are various technologies, such as Microsoft .NET, Ruby, Microsoft Silverlight, Python and Microsoft Office. DLR allows data to be accessed amongst various technologies and languages with the help of appropriate binders. Now, let's discuss about the three features, **Expression Trees**, **Call Site Caching**, and **Dynamic Object Interoperability** of DLR, which are as follows:

- **Expression Trees:** Refers to the representation of code in a data structure similar to a tree. However, expression trees in DLR, are the advanced version of the expression trees that were introduced with LINQ in .NET 2008. Expression trees are used to represent the language semantics in DLR; therefore, DLR has extended the LINQ expression trees to incorporate control flow, assignment, and various other language-modeling nodes. If you are using any .NET language that uses DLR, then, the language needs to produce expression trees and not the Intermediate Language (IL), which is generated by DLR automatically.
- **Call Site Caching:** Refers to a place in your source code wherein certain operations, such as a-b are performed. When you perform the operation a-b, DLR usually caches the characteristics of a and b and all the requisite information about the operation. If the operation that you are currently performing has ever been performed earlier, then, DLR simply retrieves all the essential information from the cache for faster dispatch.
- **Dynamic Object Interoperability:** Implies that the dynamic behavioral abilities are not automatically added to all objects and .NET types. An object can have dynamic capabilities, only if its corresponding class has dynamic dispatch capabilities. A class that implements the **IDynamicObject** interface or derives from the **DynamicObject** class and overrides a few methods of the **DynamicObject** class provides dynamic dispatch capabilities.

Binders in call sites are used by DLR to communicate with .NET Framework and various other services, such as Silverlight and COM. Binders actually represent language-specific semantics and specify how a particular operation can be performed at the call site. All the dynamically and statically typed languages can use DLR for the purpose of sharing libraries and gaining access to all the technologies that DLR supports. The functionality of each of the binders (Fig.C#-11.1) is as follows:

- **Object binder:** Enables to talk to .NET objects
- **JavaScript binder:** Enables to talk to JavaScript in Silverlight
- **IronPython binder:** Enables to talk to IronPython
- **IronRuby binder:** Enables to talk to IronRuby
- **COM binder:** Enables to talk to COM

Now, let's learn about the dynamic type in the next section.

Exploring the Dynamic Type

C# 4.0 has introduced a number of new features and one amongst them is the new static type called **dynamic**. When you use a **dynamic** type, it indicates to the compiler that its type checking should be performed dynamically at runtime and not during compilation. Microsoft has designed a C# runtime binder to perform late binding of types that have been declared as **dynamic**.

The syntax to declare a variable as dynamic is as follows:

```
dynamic d;
```

C# has a **var** type, which is similar to dynamic type but there is a difference between dynamic type and **var**. When you declare a variable using the **var** keyword, its type is resolved during compilation. At the time of compilation the compiler identifies the data type of value that is assigned to a variable. Moreover, Visual Studio also provides IntelliSense for its methods and properties. However, when you declare a variable as dynamic, the compiler has no idea about the data type of value that is assigned to a variable. Furthermore, if you have a variable of dynamic type then, its type can be changed during runtime without any chances of encountering an error.

Now, let's create a console application named **DynamicType** to see that a variable declared as **dynamic** can hold any type of data and that its type can be altered during runtime by performing the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open the Visual Studio 2010 IDE.
2. *Select* **File→New→Project** from menu bar of the Visual Studio 2010 IDE to open the **New Project** dialog box.
3. *Select* **Visual C#** in the **Installed Templates** pane and then *select* **Console Application** option from the middle pane of the **New Project** dialog box.
4. *Enter* **DynamicType** in the **Name** text box to specify the name of the application and *specify* an appropriate location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **c:\users\temp\documents\visual studio 2010\Projects**.
5. *Click* the **OK** button. The **DynamicType** application is created.
6. *Add* the highlighted code given in Listing 11.1 in the **Program.cs** file:

Listing 11.1: Showing the Code to Create the Dynamic Type

```
namespace DynamicType
{
 public class Program
 {
  dynamic d1 = 5;
  dynamic d2 = "Hello";
  dynamic d3 = DateTime.Now;
  dynamic d4 = 123;
  public static void Main(string[] args)
  {
   Program p= new Program();
   Console.WriteLine("Values a dynamic type can hold");
   Console.WriteLine(p.d1);
   Console.WriteLine(p.d2);
   Console.WriteLine(p.d3);
   Console.WriteLine("Changing type during runtime");
   Console.WriteLine(p.d4.GetType());
   Console.WriteLine(p.d4);
   p.d4 = "I am a dynamic type";
   Console.WriteLine(p.d4.GetType());
   Console.WriteLine(p.d4);
   p.d4 = new Program();
   Console.WriteLine(p.d4.GetType());
   Console.ReadLine();
  }
 }
}
```

In Listing 11.1, you can see that a dynamic variable can hold any type of data, such as integer, string, and DateTime type. Also note that we have declared a dynamic variable **d4** that has been initially assigned an integer value. During the execution of this program, you notice that the type of **d4** changes from **System.Int32** to **System.String** and then finally to **Program** type.

7. *Press* the **F5** key to run the **DynamicType** application. The output appears, as shown in Fig.C#-11.2:

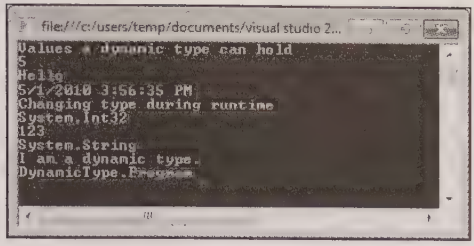


Fig.C#-11.2

In Fig.C#-11.2, the type of the dynamic variable **d4** changes dynamically and no error occurs. You can notice that the **Console.WriteLine()** statements display the runtime type of the **d4** dynamic type.

You have already learned that the compile time type checking of a dynamic type is bypassed because a dynamic type behaves similar to an object type. The type checking is bypassed for a dynamic type during compile time because at compile time the behavior of a dynamic type is similar to that of an object type. During the runtime, data type of a dynamic variable is inferred according to the value assigned to the dynamic variable. However, object types are not to be confused with dynamic types. Consider the following code snippet to differentiate the object and dynamic type:

```
dynamic dynVar = 1;
dynVar = dynVar + 5;
object objVar = 3;
objVar = objVar + 5;
```

If you compile the preceding code snippet, an error occurs at line number 3, where you try to add 3 to an object type. However, no error is encountered when you add 5 to the dynamic variable **dynVar**. In fact, the statement where you have added 5 to the dynamic variable is not checked during compilation. Also note that the result of most dynamic operations is always dynamic.

Now that you have learned about dynamic types, let's proceed further and learn about implicit conversion into a dynamic type.

Implicit Conversion into Dynamic Type

You already know that when we convert a variable of one data type into a variable of another data type, then the process is known as type casting. You also know that when the compiler converts a data type into another type of data automatically, it is known as implicit conversion. You can assign any data type to a dynamic type variable to switch between dynamic and non-dynamic behavior. The following code snippet shows implicit conversion of a data type into a dynamic type:

```
dynamic dynVar1 = 5;
dynamic dynVar2 = "This is a string";
```

By using the keyword, dynamic you can add dynamic behavior to a variable. However, you can also remove the dynamic behavior of a dynamic type by converting it into some other data type. You can dynamically implement implicit conversion from dynamic type to any type of your choice, using constructs similar to assignment statement, as shown in the following code snippet:

```
dynamic d = 7;
int a = d;
```

To run the preceding code in your program, you need to assign the dynamic type d into the integer variable a inside a constructor for your class.

Now, let's learn more about dynamic operations in the following section.

Dynamic Operations

You have learned that when you perform operations that include dynamic types, their type checking during compilation is bypassed, instead the types are resolved during runtime; therefore, the operations that include dynamic types are known as dynamic operations.

Using the dynamic type, you can invoke methods and also perform field and property accesses, indexer, and operator calls. Delegates can also be invoked dynamically using the dynamic type. Now, let's create a console application named **DynamicOperationsExample** to create and use a dynamic method, local variable, and a dynamic object by performing the following steps:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **DynamicType** application in the **Exploring Dynamic Type** section of this chapter.
2. *Enter* **DynamicOperationsExample** in the **Name** text box to specify the name of the application and specify an appropriate location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **C:\Users\temp\documents\visual studio 2010\Projects.**
3. *Click* the **OK** button. The **DynamicOperationsExample** application is created.
4. *Add* the highlighted code given in Listing 11.2 in the **Program.cs** file:

Listing 11.2: Demonstrating Dynamic Operations

```
namespace DynamicOperationsExample
{
 public class DynamicClass
{
        public dynamic SomeMethod(dynamic dVar)
        {
            dynamic dynVar = "This is a local variable";
            int num = 10;
            if (dVar is int)
            {
                return dynVar;
            }
            else
                return num;
        }
        public static void Main(string[] args)
        {
            DynamicClass dc = new DynamicClass();
            Console.WriteLine(dc.SomeMethod(20));
            Console.WriteLine(dc.SomeMethod("Dynamic Value"));
            dynamic dynObj = new DynamicClass();
            Console.WriteLine(dynObj.SomeMethod(25));
            Console.ReadLine();
        }
    }
}
```

In Listing 11.2, we have declared a dynamic method called **SomeMethod()**, which accepts a dynamic parameter type and returns a dynamic value. We have also declared a dynamic local variable inside the **SomeMethod()** method. Inside the **Main()** method, we have created a dynamic object called **dynObj**. The **SomeMethod()** method is called within the **Main()** method; therefore, the compile time type checking is bypassed and all the values are resolved at runtime.

Note

The purpose of **is** *operator is to check whether an object is compatible with a given type or not.*

5. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-11.3:

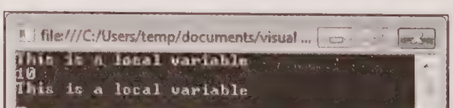


Fig.C#-11.3

In Fig.C#-11.3, you can see that when the dynamic method, **SomeMethod()** is called, a value is returned according to the type of arguments passed to the **SomeMethod()** method. Also note that the result of any dynamic operation is of dynamic type.

Now, let's examine how the dynamic types are actually resolved into the .NET types at runtime next.

Runtime Lookup for Dynamic Type

When a variable is declared as dynamic the C# compiler treats all the operations performed on that variable as dynamic. Whether it is a dynamic method invocation, property access, or indexer call, the compiler knows that the result of dynamic operations will also be dynamic; therefore, the compiler transforms all the dynamic operations into a dynamic call site or DLR call site. A DLR call site is instantiated with a field of type T, which is an object of delegate type. This delegate type is used to contain the DLR caching mechanism that has been discussed in the **Describing the Dynamic Language Runtime** section of this chapter. After creating the DLR call site, the compiler invokes the delegate, which causes DLR to perform interoperability and caching. In case you do not have an **IDynamicObject**, the **CallSiteBinder** is invoked. C# has its own derived **CallSiteBinders**, which perform the correct binding. The binding results in an expression tree, which is consolidated into the DLR call site's target delegate for DLR caching purposes. Once the expression tree becomes a part of the DLR cache mechanism, it is invoked so that the output of a user's dynamic binding can be executed.

Now, let's learn to create instances of **DynamicObject** and **ExpandoObjects** classes in the next section.

Creating the DynamicObject and ExpandoObject Class Objects

If you wish to create your own dynamic objects, then you can derive the **DynamicObject** class or simply use the **ExpandoObject** class. Using the **DynamicObject** class, you can specify dynamic behavior for an object. However, **DynamicObject** class cannot be directly instantiated. You first need to inherit this class. The **DynamicObject** class can be found in the **System.Dynamic** namespace. Using the **DynamicObject** class, you can specify the operations that can be performed on dynamic objects and how the operations will be performed. Some of the noteworthy methods of the **DynamicObject** class are enlisted in Table 11.1:

Table 11.1: Noteworthy Methods of DynamicObject Class

Value	Description
TryBinaryOperation	Specifies how the binary operations, such as addition, subtraction, division, and multiplication can be implemented in classes that inherit the DynamicObject class. Overriding this method helps in specifying the dynamic behavior for binary operations.
TryConvert	Specifies how type conversion operations are implemented. Overriding this method in the classes that inherit the DynamicObject class helps in specifying the dynamic behavior for operations that deal with converting an object of one type to an object of another type.
TryGetIndex	Specifies how those operations, which get a value by index are implemented. Overriding this method in the classes that inherit the DynamicObject class helps in specifying the dynamic behavior for indexing operations.
TryGetMember	Specifies how the operations, which get member values are implemented. Overriding this method in a class that inherits the DynamicObject class helps in specifying the dynamic behavior for operations, such as retrieving a value for a property.
TryInvoke	Specifies how the operations, which invoke an object are implemented. Overriding this method in a class that inherits the DynamicObject class helps in specifying the dynamic behavior for operations, such as invoking a delegate.
TryInvokeMember	Specifies how the operations, which invoke a member are implemented. Overriding this method in a class that inherits the DynamicObject class helps in specifying the dynamic behavior for operations in which a method is invoked.
TrySetIndex	Specifies how those operations, which set a value by index are implemented. Overriding this method in a class that inherits the DynamicObject class helps in specifying the dynamic behavior for such operations, which use an index to access an object.

Table 11.1: Noteworthy Methods of DynamicObject Class

Value	Description
TrySetMember	Specifies how the operations, which set member values are implemented. Overriding this method in a class that inherits the DynamicObject class helps in specifying the dynamic behavior for such operations, which are used to set a value for a property.
TryUnaryOperation	Specifies how unary operations are implemented. Overriding this method in a class that inherits the DynamicObject class helps in specifying the dynamic behavior for unary operations, which include increment, decrement, and many more.

You can implement the dynamic behavior to your objects by overriding the methods of the **DynamicObject** class. For instance, if you need to perform binary operations, you can override the **TryBinaryOperation** method. Now, let's create a console application named **DynamicObjectExample** to create a dynamic object that overrides the **TryGetMember()** method; therefore, whenever you access a property, dynamic object returns the name of property as a string by performing the following steps:

1. *Repeat* steps 1 to 3 as discussed earlier while creating the **DynamicType** application in the **Exploring Dynamic Type** section of this chapter.
2. *Enter* **DynamicObjectExample** in the **Name** text box to specify the name of the application and specify an appropriate location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **c:\users\temp\documents\visual studio 2010\Projects.**
3. *Click* the **OK** button. The **DynamicObjectExample** application is created.
4. *Add* the highlighted code given in Listing 11.3 in the **Program.cs** file:

Listing 11.3: Creating a Dynamic Object

```
namespace DynamicObjectExample
{
    public class MyDynamicObject : DynamicObject
    {
        public override bool TryGetMember(GetMemberBinder binder, out object result)
        {
            result = binder.Name;
            return true;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            dynamic dObj = new MyDynamicObject();
            Console.WriteLine(dObj.MyDynamicProperty);
            Console.ReadLine();
        }
    }
}
```

In Listing 11.3, we have used the **TryGetMember()** method defined in the **DynamicObject** class. The **TryGetMember()** method contains two parameters, **binder** and **result**. The **binder.Name** property of the **TryGetMember()** method returns the name of the member in which the dynamic operation is performed. In this case, the **binder.Name** property, returns **MyDynamicProperty**. The result parameter retrieves the output of the **Get** operation. Note that when **dObj.MyDynamicProperty** is called, DLR uses the C# language binder to search for a static definition of this property in the **MyDynamicObject** class. If no such property is found, DLR calls the **TryGetmember()** method of the **DynamicObject** class. This method retrieves information regarding the operation that was invoked through the binder parameter. If the dynamic operation is successful, the **TryGetMember()** method returns true. However, you must assign the actual result of the operation to the parameter result.

The syntax of **TryGetMember()** method is as follows:

```
public virtual bool TryGetMember(
```

```
    GetMemberBinder binder,
    out Object result
)
```

In the preceding syntax, **binder** provides information about the object that has invoked the dynamic operation.

5. *Press* the **F5** key and run the application. The output appears, as shown in Fig.C#-11.4:

Fig.C#-11.4

You can see in Fig.C#-11.4, the **TryGetMember()** method returns true, however, **dObj.MyDynamicProperty** returns **MyDynamicProperty**.

The **ExpandoObject** class represents an object whose members can be added and removed at runtime. Using the **ExpandoObject** class, you can use the standard syntax, such as **myObject.myMember**. You can also create an object of the **ExpandoObject** class in C# and pass it on to a function of a dynamic language, such as IronRuby, since the **ExpandoObject** class implements the **IDynamicMetaObjectProvider** interface. You can enable late binding in an object of the **ExpandoObject** class using the **dynamic** keyword. The syntax to create an instance of the **ExpandoObject** class is as follows:

```
dynamic myExpandoObj = new ExpandoObject();
```

You can also pass the instances of **ExpandoObject** class as parameters. These are treated as dynamic objects in C#. In case you try to access a method member that does not exist, using the instance of **ExpandoObject**, then, you do not get a compiler error. Instead, an exception is thrown at runtime. Now, let's create a console application named **ExpandoObjectExample** to create an instance of **ExpandoObject** and pass it as a parameter by performing the following steps:

1. *Repeat* steps 1 to 3 as discussed earlier while creating **DynamicType** application in the **Exploring Dynamic Type** section of this chapter.
2. *Enter* **ExpandoObjectExample** in the **Name** text box to specify the name of the application and *specify* an appropriate location in the **Location** combo box to save the application. In this case, we have selected the default location, which is **c:\users\temp\documents\visual studio 2010\Projects**.
3. *Click* the **OK** button. The **ExpandoObjectExample** application is created.
4. *Add* the highlighted code given in Listing 11.4 in the **Program.cs** file:

Listing 11.4: Creating the Instances of ExpandoObject Class

```
namespace ExpandoObjectExample
{
    class Program
    {
        static void Main(string[] args)
        {
            dynamic day = new ExpandoObject();
            day.Name = "Monday";
            day.DayOfWeek = "First";
            dynamic day2 = new ExpandoObject();
            day2.Name = "Tuesday";
            day2.DayOfWeek = "Second";
            WriteDay(day);
            WriteDay(day2);
            Console.ReadLine();
        }
        private static void WriteDay(dynamic Weekday)
        {
            Console.WriteLine("{0} is the {1} day of week", Weekday.Name,
          Weekday.DayOfWeek);
```

```
            }
        }
    }
```

In Listing 11.4, we have created two **ExpandoObject** class objects, **day** and **day2**. We have also defined a method called **WriteDay()** that takes an instance of **ExpandoObject** as a parameter and prints the value of the properties, **Name** and **DayOfWeek**.

5. *Press* the **F5** key to run the application. The output appears, as shown in Fig.C#-11.5:

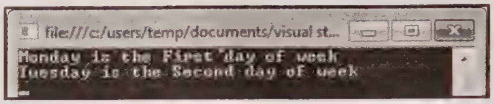


Fig.C#-11.5

In Fig.C#-11.5, the **day** and **day2** objects of **ExpandoObject** class display the value of properties, **Name** and **DayOfWeek** respectively.

Interoperating with Dynamic Languages

DLR made it simpler for language designers to develop a language that has interoperability with other DLR languages such as IronRuby and IronPython and .NET languages such as C# and VB.NET. In addition, the aim was to provide support to access the .NET Base Class library.

The theme of C# 2010 is to interoperate with dynamic languages. This interoperability with dynamic languages is intended to facilitate the use and manipulation of an object from IronPython, IronRuby, JScript, or a standard .Net library in same way. In C#, the objects of dynamic languages are exposed using the **dynamic** keyword in such a way that the objects seem familiar to a C# programmer. You can perform the following steps to use IronPython within your C# application:

1. Write a layer of your application in IronPython
2. Use Python as the configuration language
3. Use a Python library such as feedparser
4. Put a live interpreter in the application for debugging purposes

Now, let's summarize the key points that we have learned in this chapter in the next section.

Summary

In this chapter, you learned about:

- DLR, its features and services
- Dynamic type, its implicit conversion, dynamic operation, and runtime lookup
- The **DynamicObject** and **ExpandoObject** classes and instantiating their objects
- Interoperability of .NET languages with dynamic languages

Chapter 12

Introduction to Windows Workflow Foundation

In this Chapter:

- Workflow Principles
- Components of Windows Workflow Foundation
- Enhancements to Windows Workflow
- Building a Simple Workflow Application
- Implementing Conditions in Workflows
- Using Workflows with Other Applications

Windows Workflow Foundation (WF) enables .NET developers to build., monitor, and execute workflows in a .NET application. A workflow generally refers to a graphical representation of a sequence of tasks that are performed to complete a given procedure or produce some result. Adding workflows in a .NET application enables the developers to build applications that start, stop, and wait for something to occur. For example, consider a scenario where a person's identity is to be validated before a loan can be granted to him or her. Determining whether the documents furnished by an individual are authentic or not, requires various sequential and simultaneous tasks to be performed. WF enables such type of situations to be modeled into software in the form of WF applications.

.NET Framework 3.0 introduced WF along with three other foundations, Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), and Windows CardSpace (WCS). Visual Studio 2010 has in-built templates to develop workflow-enabled applications.

This chapter begins with discussion on the principles of workflows and later explains the components of WF. The chapter also discusses the enhancements that have been made to WF 4.0. Further, you learn to develop a simple workflow application, implement conditions in workflows, and use workflows with other applications.

Let's begin discussing the principles of workflows in the next section.

Workflow Principles

The workflow platform that you use to develop workflow-based applications must incorporate certain principles, which are as follows:

- Workflows help in coordinating the work performed by people and software
- Workflows are long running and stateful
- Workflows are based on extensible models
- Workflows are transparent and dynamic throughout their lifecycle

Note

Those applications that can be easily created on Windows using WF are known as workflow-enabled applications. Workflow- based applications are same as the workflow-enabled applications.

Let's discuss each of these principles in detail next.

Coordinating the Work Performed by People and Software

As workflows allow the work done by people to be connected with work done in software, therefore; it is essential that workflows interact with people. Workflows that implement the principle of coordinating the work performed by people and software enable human interaction with any user interface (UI), such as e-mails, Web pages, mobile devices, or other front ends. WF provides the necessary infrastructure to effectively handle human interaction and all the related issues.

Long Running and Stateful

Human beings interact with software systems on an ad hoc basis. For example, they may interact with software systems after few minutes, hours, or months; therefore, workflows should be able to run for long periods. However, running a workflow for long periods, and storing a running workflow in memory is not practical due to many reasons. If every running workflow is stored in memory while waiting for something to occur, the server would run out of memory immediately. In addition, if the server crashes, the volatile memory is cleared and all data is lost.

Based on Extensible Models

As stated earlier, workflows serve the purpose of automating business processes. Now, since each type of business has a wide range of problems; therefore, a workflow platform needs to be extensible. WF provides a set of base activities, such as **IfElse**, **Code**, and **Delay**, to build a workflow. You can extend these activities or

build new activities to meet your requirements. Besides activities, you can also extend services, such as, tracking, management, and persistence, provided by the runtime engine.

Transparent and Dynamic Throughout Their Lifecycle

As WF is based on a declarative and visual design-time model, existing workflows can be modified without changing their source code.

In this section, the principles of workflows were discussed. Next, we discuss the various components of WF.

Components of Windows Workflow Foundation

WF consists of several components, such as workflow, activity, activity library, runtime engine, runtime services, and host process that work with an application to perform the desired workflow. Fig.C#-12.1 shows the architecture of WF along with its components:

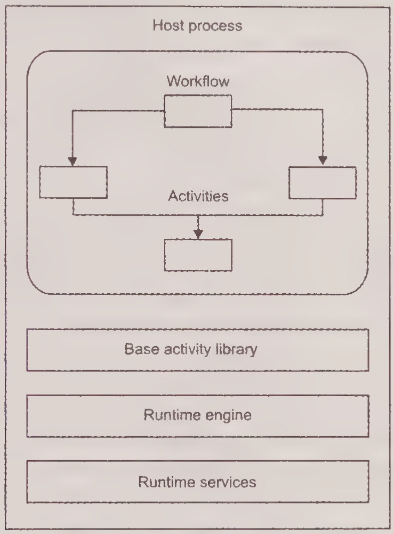


Fig.C#-12.1

Following are the six major components of WF architecture:

- Workflow
- Activity
- Base activity library
- Host process
- Runtime engine
- Runtime services

Let's discuss each of these components next.

Workflow and its Types

A workflow can be understood as a declarative program wherein each program statement is represented in terms of a component, called an activity. In other words, a workflow defines the business logic on which a program is based. A business process can involve applications as well as people. Workflows that are developed to automate interactions among applications are known as system workflows. Such workflows are usually static and predictable. In contrast to system workflows, workflows that are intended to coordinate interactions among people are known as human workflows. Applications that involve human interactions generally need more flexibility than others, because people may change their minds, introduce new ideas and exceptions, and cancel

a process unexpectedly. Though there are differences between system and human workflows, however; WF supports both system and human workflows in a unified manner.

In WF 4.0, two types of workflows are used. These types of workflows are given as follows:

- **Flowchart workflows:** Helps you to create workflows using the common flowchart elements.
- **Procedural workflows:** Helps you to create workflows using basic and sequential execution standards.

Flowchart Workflows

Flowchart is a common standard to design programs. In WF, the Flowchart activity is generally used to implement a non-sequential workflow, and occasionally it implements sequential workflows incase FlowDecision nodes are not used. The Flowchart activity contains a collection of flow nodes, which inherit from the FlowNode class. The following types of nodes or elements can be a part of a flowchart:

- **FlowStep:** Shows one step of execution in the flowchart
- **FlowDecision:** Shows the execution on the basis of a Boolean condition. It is as similar to the **If** construct.
- **FlowSwitch:** Shows the execution on the basis of an exclusive switch. It is as similar to the **Switch** construct.

Next, let's discuss about the procedural workflows.

Procedural Workflows

In WF, procedural workflows use flow control constructs, such as **While**, **Switch**, **ForEach**, and **If**. These flow control constructs are as similar to those found in procedural languages. Procedural workflows can consist of other flow control activities also, such as **Flowchart** and **Sequence**.

Next, let's learn about the activity component in WF.

Activity

In Windows Workflow Foundation 4.0, an activity is the basic unit of composition and execution of a workflow. This in other words means that an activity represents a unit of work in Windows Workflow Foundation 4.0. Each activity in a workflow consists of its own variables and arguments and is a subclass of the Activity class. The variables in an activity are used to store the data in an activity while the arguments in an activity are used to move data in and out of an activity.

Workflows in Windows Workflow Foundation 4.0 can be created by implementing the **CodeActivity** abstract class for basic custom activity functionality, such as implementing control flow, conditions, event handling, and state management. In addition, the **NativeActivity** abstract class can be implemented to develop custom activities that implement specific behavior from scratch.

Next, let's learn about the Activity Library in WF.

Expanded In-Built Base Activity Library

The base activity library is a collection of activities that are used to create workflows. Activities are the basic building blocks for workflows. The action that an activity implements can be a simple action, such as a delay or a composite activity, which is composed of multiple child activities. The activities contained in the base activity library provide functionalities for control flow, state management, event handling, and communicating with applications and services.

Table 12.1 lists some important activities available in the base activity library:

Table 12.1: Some Important Activities Available in the Base Activity Library

Activity	Description
`CodeActivity`	Enables you to add Visual C# code to your workflow.
`ConditionedActivityGroup`	Contains a collection of child activities whose execution order is based on certain conditions.

Table 12.1: Some Important Activities Available in the Base Activity Library	
Activity	**Description**
DelayActivity	Pauses the flow of a workflow for a specified time period.
EventDrivenActivity	Defines one or more activities that are executed when a specific event occurs.
EventHandlersActivity	Allows you to associate an event with an activity. This is a composite activity that contains a collection of event handlers.
EventHandlingScopeActivity	Executes its main child activity along with an EventHandlersActivity activity.
FaultHandlerActivity	Handles an exception of a specified type.
FaultHandlersActivity	Represents a composite activity that has an ordered list of child activities of the FaultHandlerActivity type.
IfElseActivity	Allows you to execute two or more different workflow paths based on a condition.
IfElseBranchActivity	Represents a branch (path) of an IfElseActivity activity.
InvokeWebServiceActivity	Enables a workflow to invoke a Web service.
InvokeWorkflowActivity	Enables a workflow to invoke another workflow.
ListenActivity	Makes the workflow wait for any one of several possible events before the activity proceeds. The ListenActivity activity represents a composite activity that contains only EventDrivenActivity activities.
PolicyActivity	Represents a collection of rules. A rule contains conditions and resulting actions.
ReplicatorActivity	Creates multiple instances of a single child activity.
SequenceActivity	Executes a group of activities at a time in a defined order.
SetStateActivity	Changes the state of the state machine of a workflow. In other words, it makes a transition from one state to another state.
StateActivity	Represents a state in the state machine of a workflow.
StateFinalizationActivity	Defines one or more activities that are executed by default when there is a transition from one state to another state in a state machine workflow.
StateInitializationActivity	Defines one or more activities that are executed by default when a specific state is entered.
SuspendActivity	Suspends the operation of your workflow temporarily.
SynchronizationScopeActivity	Executes child activities that it contains sequentially in a synchronized domain.
TerminateActivity	Enables you to immediately finish the operation of your workflow in case of an error condition.
ThrowActivity	Raises exceptions declaratively, usually in response to exceptional conditions detected in a workflow.
WebServiceFaultActivity	Enables a fault to be sent to a Web service client from a workflow. WebServiceFaultActivity activity is equivalent to throwing an exception in an ASMX Web service framework method.
WebServiceInputActivity	Receives data from a Web service.
WebServiceOutputActivity	Responds to a Web service request made to a workflow.
WhileActivity	Executes one or more activities repeatedly until a condition is true.

In Table 12.1, the activities that you can use in your Workflow applications are enlisted.

Host Process

Since a workflow created with WF is not a standalone product; therefore, it needs a host application to be hosted and run. Host process is the process within which a workflow is hosted and run. A host process may be a Windows Forms application, a Web application, or a Web service application, where a user interaction can also take place.

Runtime Engine

A runtime engine is not a separate service or process; it runs within the host process and is responsible to manage and execute each workflow instance. A host process may have multiple runtime engines running concurrently where each engine can execute multiple workflow instances simultaneously.

Runtime Services

Runtime services consist of predefined and user-defined classes that interact with system outside the workflow. To run a particular instance of a workflow, the workflow runtime depends on a number of runtime services. Custom implementations of the runtime services can also be developed to meet the specific needs of an application. The runtime services are defined within the **System.Workflow.Runtime.Hosting** namespace. The following are important runtime services provided by WF:

- **Persistence service:** Enables you to save the state of a workflow for later use. You can restart the workflow as per the requirement, even after weeks of inactivity.
- **Tracking service:** Enables developers to monitor the state of workflows. This is particularly useful when you have multiple active workflows at a time (for example, a shopping cart application).
- **Transactions services:** Enables transaction support to maintain data integrity.

In this section, we have discussed the various components of WF. Next, let's examine some of the enhancements to WF.

Enhancements to Windows Workflow

WF 4.0 includes several changes as compared to the previous version of WF that help to easily create, execute, and maintain Workflow applications. Following are some of the improvements in WF 4.0:

- Uses activity as the basic unit of creating a workflow instead of using classes such as SequentialWorkflowActivity or StatemachineWorkflowActivity. The **CodeActivity** class can be implemented to perform basic activity functionality. However, to customize the basic activity functionality, you should use the **NativeActivity** class.
- Introduces a new control flow activity called the **Flowchart**, which provides an event-driven programming model.
- Introduces the following new features to the built-in activity library:
  - Introduces new control flow activities, such as **DoWhile**, **Pick**, **TryCatch**, **ForEach**, **Switch**, and **ParallelForEach**.
  - Adds activities to the built-in activity library, such as **Assign** and **AddToCollection** activities to help manipulate data members.
  - Introduces new messaging activities such as **SendContent** and **ReceiveReply**.
  - Adds activities to the built-in activity library, such as **TransactionScope** and **Compensate** to control transactions.
- Includes the following new options to run workflows:
  - **WorkflowServiceHost:** Specifies a host for workflows.
  - **WorkflowApplication:** Specifies a host for a particular instance of a workflow.
  - **WorkflowInvoker**: Acts as a method call and invokes a workflow.

- Introduces the **Bookmark** object that can be used for resuming a pending workflow.

Now, let's learn to create a simple workflow application in the next section.

Building a Simple Workflow Application

A workflow application is similar to a Windows Forms application and can be developed by performing similar steps that are required to develop a Windows Forms application. Let's learn to develop a simple workflow application that accepts a string and reverses it by performing the following steps:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to start Visual Studio 2010.
2. *Select* **File→New→Project** on menu bar or *press* the **CTRL+SHIFT+N** keys. The **New Project** dialog box opens.
3. *Select* **Visual C#→Workflow** in the **Installed Templates** pane.
4. *Select* the **Workflow Console Application** option from the middle pane.
5. *Enter* a name for your application in the **Name** text box. In this case, we have entered **ASimpleWorkflowApplication.**
6. *Enter* the complete path of the folder where you want to save your application in the **Location** combo box. In this case, we have selected the location as **D:\Books\Simple Steps\C# 2010 SS\Applications\Chapter 12.**
7. *Click* the **OK** button.

   The **New Project** dialog box is closed and a new Workflow Console application named **ASimpleWorkflowApplication** is created, as shown in Fig.C#-12.2:

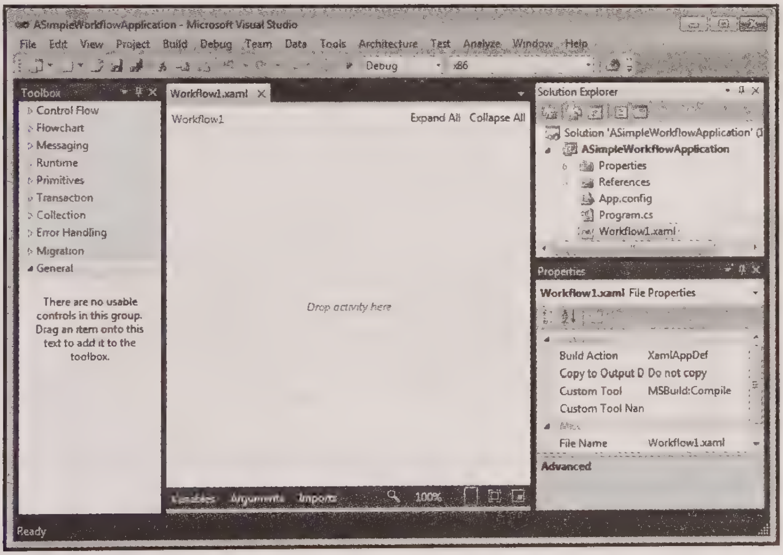


Fig:C#-12.2

   In Fig.C#-12.2, three files have been automatically created. These are the application configuration file called **App.config**, the **Program.cs** file that contains the **Main()** method, and the **Workflow1.xaml** Workflow file.

8. *Drag* a **WriteLine** activity from the **Primitives** tab in **Toolbox** and *drop* it on the WF designer, as shown in Fig.C#-12.3:

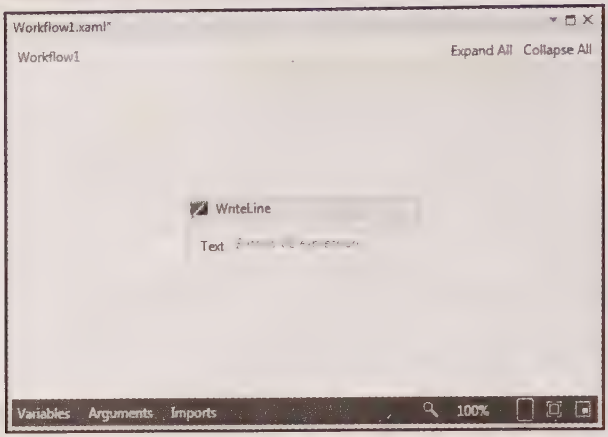


Fig.C#-12.3

The **WriteLine** activity is used to display a line of text.

9. *Select* the **WriteLine** activity and open its **Properties** window.

The **Properties** window appears, as shown in Fig.C#-12.4:

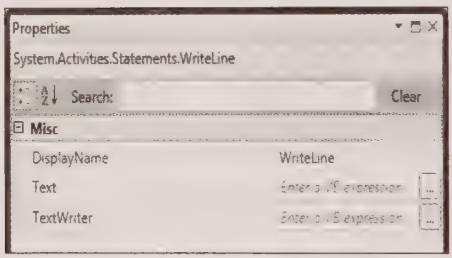


Fig.C#-12.4

In Fig.C#-12.4, the **WriteLine** activity has three properties, **DisplayName**, **Text,** and **TextWriter**. The **DisplayName** property is the name of the **WriteLine** activity displayed in the WF designer. In the **Text** property, you can enter the text expression that is displayed when the application is run. Note that the text expression must be enclosed within double quotes. In addition, your text must be a valid Visual Basic statement even if you are creating a C# project. The **TextWriter** property is optional and is used to represent a text expression.

10. *Enter* the text, "**This is a Workflow Application**" in the **Text** property of the **WriteLine** activity in **Properties** window, as shown in Fig.C#-12.5:

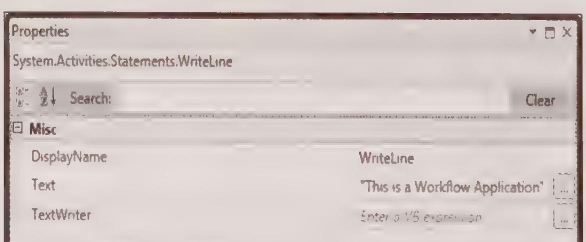


Fig.C#-12.5

11. *Press* the **F5** key to run the **ASimpleWorkflowApplication** Workflow application. The output appears, as shown in Fig.C#-12.6:

Fig.C#-12.6

In this section, you have learned to develop a simple workflow application. Next, you learn to implement conditions in workflows.

Implementing Conditions in Workflows

Similar to a Visual C# program, you can also implement conditions in workflow that you create with WF. Activities, such as **If** and **While**, can be used to implement a condition in a workflow application. You can implement a simple condition by using one of the following ways:

- **By creating a rule condition:** Creates conditions either directly in code or using a tool, called the **Rule Condition Editor**. Rule conditions are stored in a separate Extensible Markup Language (XML) file. When a rule condition is encountered in a workflow, the expression in a condition is evaluated and a **Boolean** value is returned.
- **By creating a code condition:** Refers to directly defining a condition in code. A code condition can be created by writing a method in the code. The method contains code for the condition and returns a **Boolean** value. Now, when the workflow is executed and the condition is encountered, the method that contains code for the condition is called. In this process, the value returned by the method is used as the result for the code condition.

Let's learn to implement conditions in a workflow application by performing the following steps:

1. *Create* a new **Workflow Console Application** with the name **ImplementingConditions**.
2. *Drag* a **Sequence** activity from the **Control Flow** tab of **Toolbox** and *drop* it on the Workflow design view (Fig.C#-12.7).
3. *Select* the **Sequence** activity and *click* the **Variables** button at the bottom of the Workflow design view, as shown in Fig.C#-12.7:

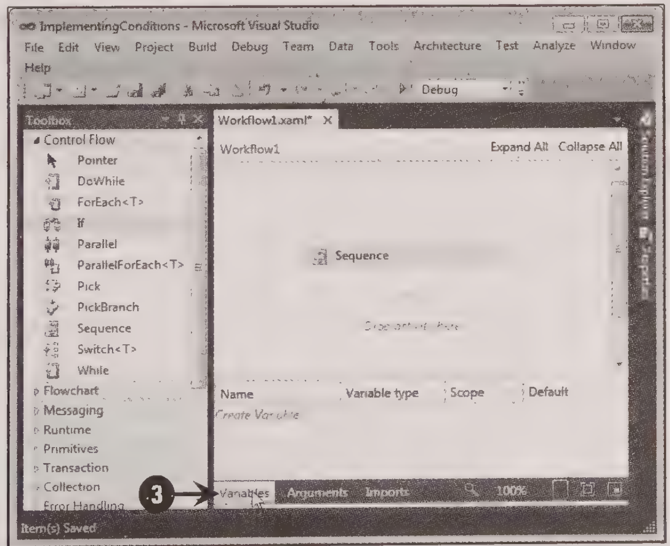


Fig.C#-12.7

A workflow is similar to a program that you create; therefore, it is capable of storing variables, passing arguments or data in and out of workflow, and importing any namespaces that you might need. You can declare and define variables for your Workflow application using the **Variables** button, pass arguments using the **Arguments** button and import namespaces using the **Imports** button present at the bottom of the Workflow design view.

4 *Add* a variable called **strHello** (Fig.C#-12.8).

5 *Enter* **"Hello World"** under the **Default** tab, as shown in Fig.C#-12.8:

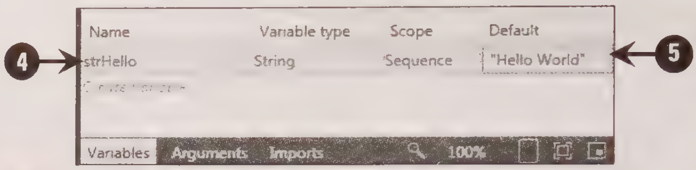


Fig.C#-12.8

In Fig.C#-12.8, the **Variable type** and **Scope** for **strHello** are already set to **String** and **Sequence** respectively; therefore, you do not need to change them.

6 *Click* the **Arguments** button to *declare* an argument called **strName** for accepting some input from the user, as shown in Fig.C#-12.9:

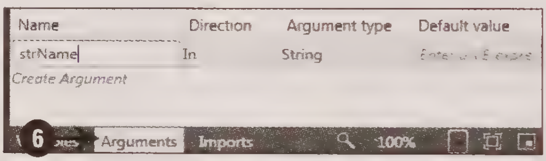


Fig.C#-12.9

In Fig.C#-12.9, the **Direction** and **Argument type** are already set to **In** and **String**, respectively; therefore, you do not need to change them.

7 *Drag* an **If** activity from the **Control Flow** tab of **Toolbox** and *drop* it inside the **Sequence** activity to implement the if-then-else conditional expression.

8 *Add* a **Condition** inside the **If** activity, to check whether the length of the string that user has entered is greater than 6 or not, as shown in Fig.C#-12.10:

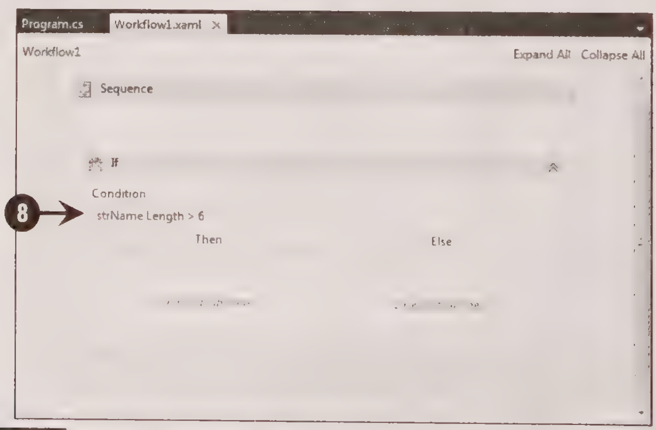


Fig.C#-12.10

9 *Drag* a **WriteLine** activity and *drop* it inside the **Then** part of the **If** activity (Fig.C#-12.11).

10 *Select* the **WriteLine** activity and *open* its **Properties** window. The **Properties** window appears (Fig.C#-12.11).

11 *Set* the **Text** property of the **WriteLine** activity to "**The length of the string is greater than 6**", as shown in Fig.C#-12.11:

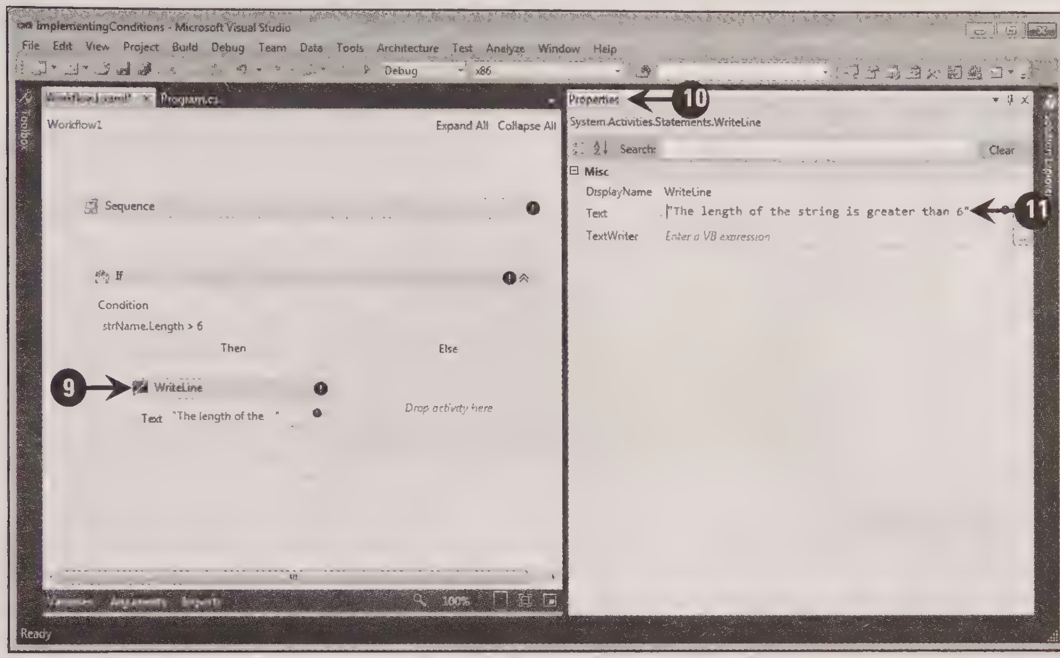


Fig.C#-12.11

12 *Drag* another **WriteLine** activity and *drop* it inside the **Else** part of the **If** activity (Fig.C#-12.12).

13 *Set* its **Text** property to "**The length of the string is less than 6**", as shown in Fig.C#-12.12:

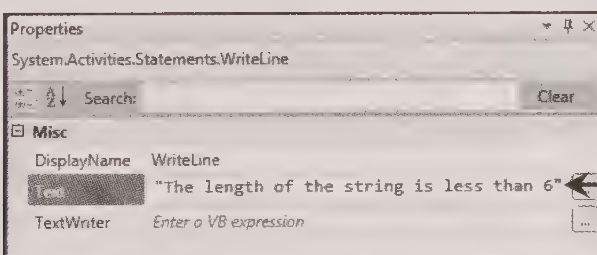


Fig.C#-12.12

14 *Add* the highlighted code given in Listing 12.1 in the **Program.cs** file:

Listing 12.1: Showing Code to Implement If-Else condition in the Program.cs File

```
namespace ImplementingConditions
{
    class Program
    {
        public static void Main(string[] args)
        {
            Workflow1 myWorkflow = new Workflow1();
```

```
            Console.WriteLine("Implementing Conditions in a Workflow");
            Console.WriteLine("Enter a Name:");
            myWorkflow.strName = Console.ReadLine();
            WorkflowInvoker.Invoke(myWorkflow);
            Console.ReadLine();
        }
    }
}
```

In Listing 12.1, the user is asked to enter a name. When the user enters a name, the **Condition** inside the **If** activity is used to check whether the length of the string is greater than 6 or not. Depending on the length of the string, the output is displayed. In addition, the **Invoke()** method of the **WorkflowInvoker** class is used to invoke the **ImplementingConditions** Workflow application.

15. *Press* the **F5** key to run the **ImplementingConditions** Workflow application. The output appears, as shown in Fig.C#-12.13:

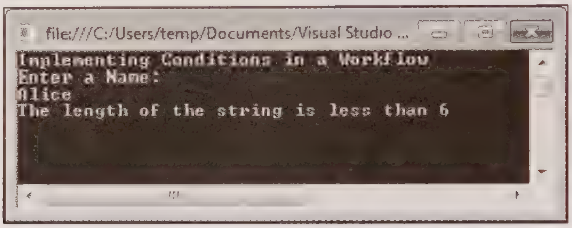


Fig.C#-12.13

In this section, you have learned to implement conditions in workflows. Next, you learn to use custom workflows with some other applications, such as a Console application.

Using Workflows with Other Applications

WF allows you to create standalone workflow applications that can later be integrated with some other applications, such as Console, Windows Forms applications, and Web applications. You can use custom workflows with a Console application by first creating an activity library that contains a custom **CodeActivity** activity class and then adding a Console application. The Console application uses the **CodeActivity** activity class created inside the activity library.

Perform the following steps to create a complete application:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to start Visual Studio 2010.
2. *Select* **File→New→Project** on menu bar or *press* the **CTRL+SHIFT+N** keys. The **New Project** dialog box opens.
3. *Select* **Visual C#→Workflow** in the **Installed Templates** pane.
4. *Select* the **Activity Library** option in the middle pane.
5. *Enter* a name for activity library in the **Name** text box. In this case, we have entered **MyCustomActivityLibrary**.
6. *Enter* the complete path of the folder where you want to save your activity library in the **Location** combo box.
7. *Click* the **OK** button.

   The **MyCustomActivityLibrary** activity library is created, as shown in Fig.C#-12.14:

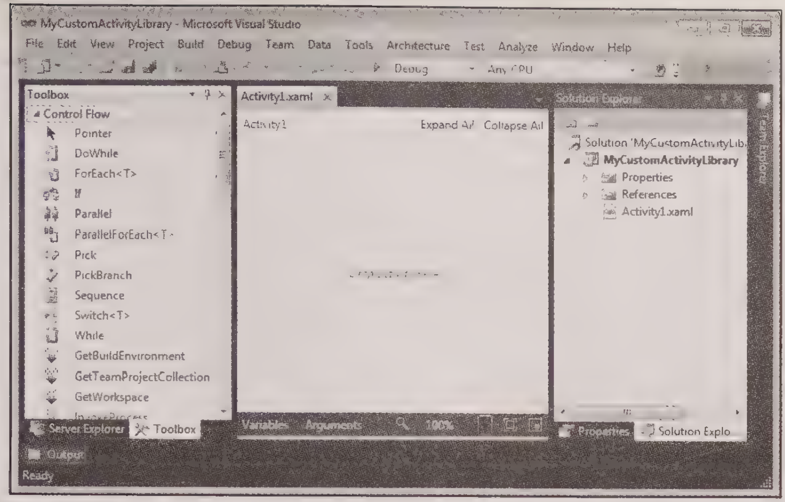


Fig.C#-12.14

In Fig.C#-12.14, you can see that **Solution Explorer** contains an **XAML** file, called **Activity1.xaml,** which is already added to the **MyCustomActivityLibrary** activity library (Fig.C#-12.14).

8. *Right-click* the **Activity1.xaml** file in **Solution Explorer** and *select* **Delete** option from the context menu that appears. The **Activity1.xaml** file is deleted because activity is created using code in our **MyCustomActivityLibrary** application.

9. *Right-click* the **MyCustomActivityLibrary** activity library in **Solution Explorer** and *select* **Add→New Item**. The **Add New Item** dialog box appears (Fig.C#-12.15).

10. *Select* **Visual C# Items→Workflow** in the **Installed Templates** pane of the **Add New Item** dialog box (Fig.C#-12.15).

11. *Select* the **Code Activity** option from middle pane (Fig.C#-12.15).

12. *Enter* a name for the **Code Activity** activity class. In this case, we have named it **MyCodeActivity.cs** (Fig.C#-12.15).

13. *Click* the **Add** button to add the **MyCodeActivity.cs** activity class to the **MyCustomActivityLibrary** activity library, as shown in Fig.C#-12.15:

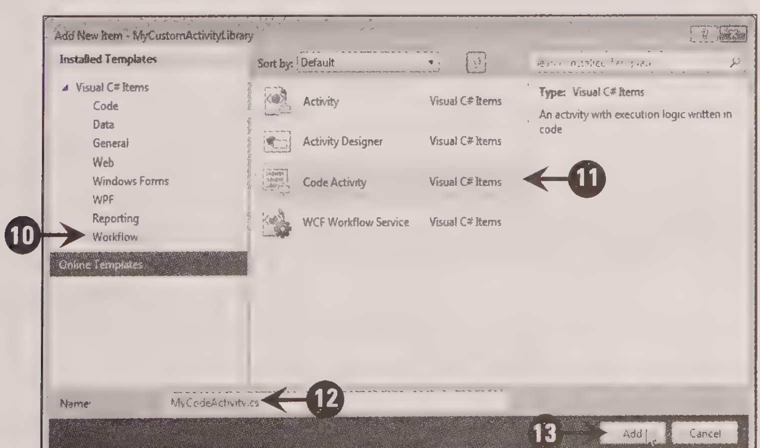


Fig.C#-12.15

The **MyCodeActivity.cs** custom activity class is added to the **MyCustomActivityLibrary** activity library.

14 *Add* the highlighted code given in Listing 12.2 to the **MyCodeActivity.cs** custom activity class:

Listing 12.2: Showing the Code to Add in the **MyCodeActivity.cs** File

```
namespace MyCustomActivityLibrary
{
    public sealed class MyCodeActivity : CodeActivity
    {
        // Define an activity input argument of type int
        InArgument<int> num1;
        public InArgument<int> Num1
        {
            get { return num1; }
            set { num1 = value; }
        }
        InArgument<int> num2;
        public InArgument<int> Num2
        {
            get { return num2; }
            set { num2 = value; }
        }
        // If your activity returns a value, derive from CodeActivity<TResult>
        // and return the value from the Execute method.
        protected override void Execute(CodeActivityContext context)
        {
            // Obtain the runtime value of the Text input argument
            int a = context.GetValue(this.Num1);
            int b = context.GetValue(this.Num2);
            int c = a + b;
            Console.WriteLine("The addition of " + a + " and " + b + ": " +
            c.ToString());
            Console.ReadLine();
        }
    }
}
```

In Listing 12.2, the **MyCodeActivity.cs** custom activity class inherits the **CodeActivity** activity class. Two input arguments, **num1** and **num2**, are added to the **MyCodeActivity.cs** custom activity class. Two properties called **Num1** and **Num2** are declared and defined to assign new values to **num1** and **num2**, respectively using the set **accessor**. The get **accessor** in the **Num1** and **Num2** properties is used to return the property value. The **Execute()** method of the **CodeActivity** activity class is overridden in the **MyCodeActivity.cs** custom activity class so that it can execute a custom activity. The **context.GetValue()** method is used inside the **Execute()** method to assign the properties, **Num1** and **Num2** to local integer variables, **a** and **b**, respectively. Finally, the result of addition of **a** and **b** is stored in the integer variable **c** and then displayed using the **Console.WriteLine()** statement.

15 *Press* the **CTRL+SHIFT+B** keys together to build the **MyCustomActivityLibrary** activity library. A .NET assembly, named **MyCustomActivityLibrary.dll** is created. You can use this assembly in other .NET applications also.

You have created a reusable .NET code library that contains a custom activity class. Now, let's create a Workflow Console Application to add the **MyCustomActivityLibrary** activity library in it.

16 *Right-click* Solution **'MyCustomActivityLibrary' (1 project)** in **Solution Explorer** (Fig.C#-12.16).

17 *Select* **Add→New Project** from the context menu that appears, as shown in Fig.C#-12.16:

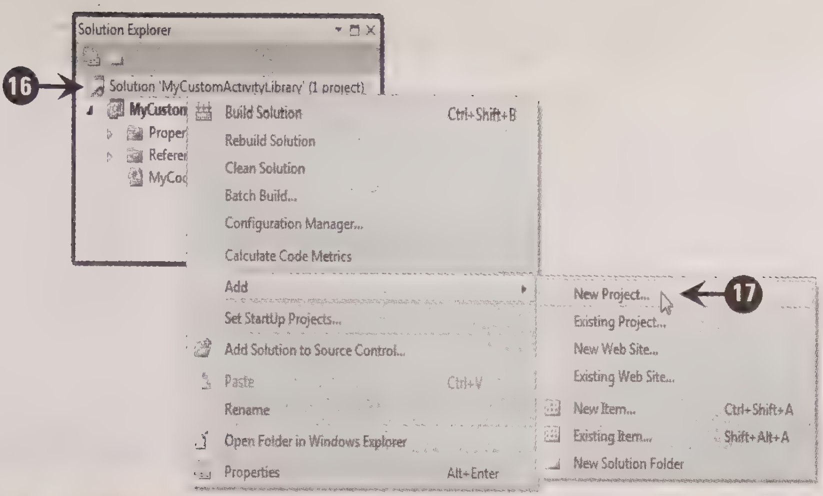


Fig.C#-12.16

The **Add New Project** dialog box appears (Fig.C#-12.17)

18 *Select* **Workflow Console Application** from the middle pane and name it as **CustomLibraryTestApplication** (Fig.C#-12.17).

19 *Click* the **OK** button to add the **CustomLibraryTestApplication** Workflow application, as shown in Fig.C#-12.17:

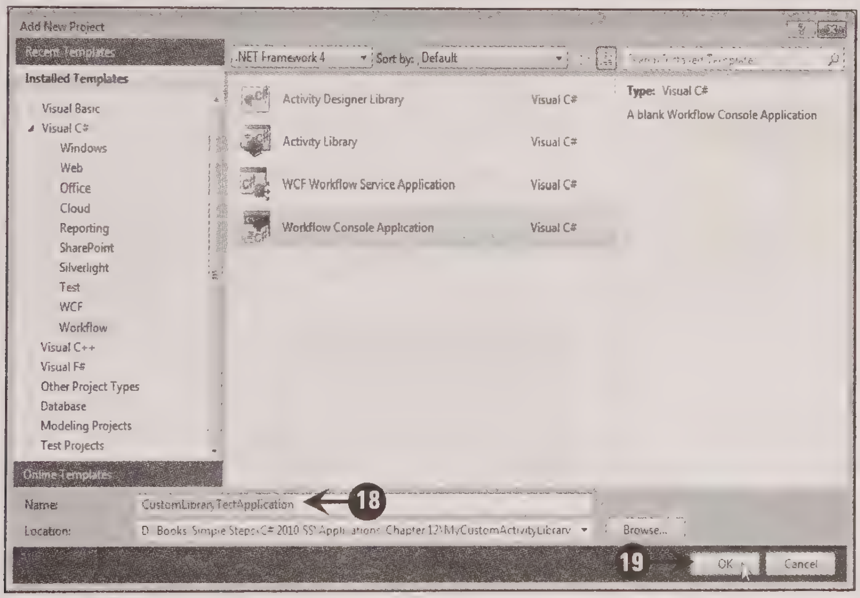


Fig.C#-12.17

20 *Right-click* the **References** option under the **CustomLibraryTestApplication** Workflow application in **Solution Explorer** and then *select* the **Add Reference** option from the context menu that appears to add a reference to the **MyCodeActivity.cs** custom activity class, as shown in Fig.C#-12.18:

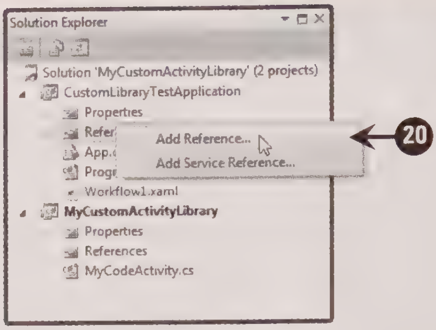


Fig.C#-12.18

The **Add Reference** dialog box appears (Fig.C#-12.19).

21 *Select* the **MyCustomActivityLibrary** option and click the **OK** button, as shown in Fig.C#-12.19:

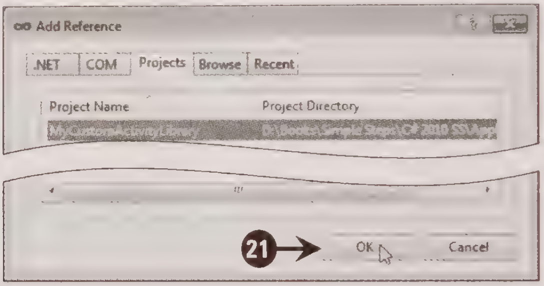


Fig.C#-12.19

In Fig.C#-12.19, a reference to the **MyCustomActivityLibrary** activity library is added to the **CustomLibraryTestApplication** Workflow application.

As the **MyCodeActivity** custom activity class and the **CustomLibraryTestApplication** Workflow application are in the same solution; therefore, the **MyCodeActivity** custom activity class is displayed in **Toolbox**, as shown in Fig.C#-12.20:

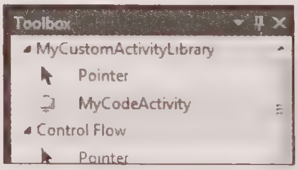


Fig.C#-12.20

22 *Drag* a **Sequence** activity from the **Control Flow** tab in **Toolbox** and drop it into the Workflow designer.

23 *Declare* the two input arguments required by the **MyCodeActivity**.cs custom activity class in the **CustomLibraryTestApplication** Workflow application, using the **Arguments** button at the bottom of the workflow designer.

24 *Click* the **Arguments** button and *add* the first argument, **num1**. *Set* the **Direction**, **Argument type**, and **Default value** properties of **num1** as **In**, **Int32**, and **170**, respectively (Fig.C#-12.21).

25 Similarly, *set* the **Name**, **Direction**, **Argument type**, and **Default value** properties of the second argument, **num2**, as shown in Fig.C#-12.21:

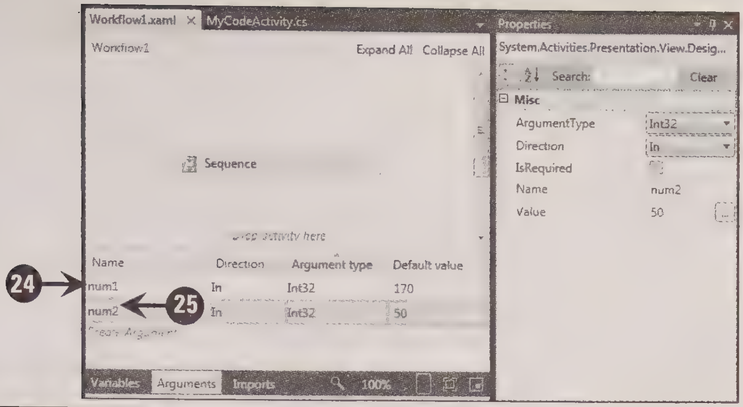


Fig.C#-12.21

26 *Drag* the **MyCodeActivity** custom activity from **Toolbox** and drop it inside the **Sequence** activity in workflow designer. The **Workflow1.xaml** workflow application appears, as shown in Fig.C#-12.22:

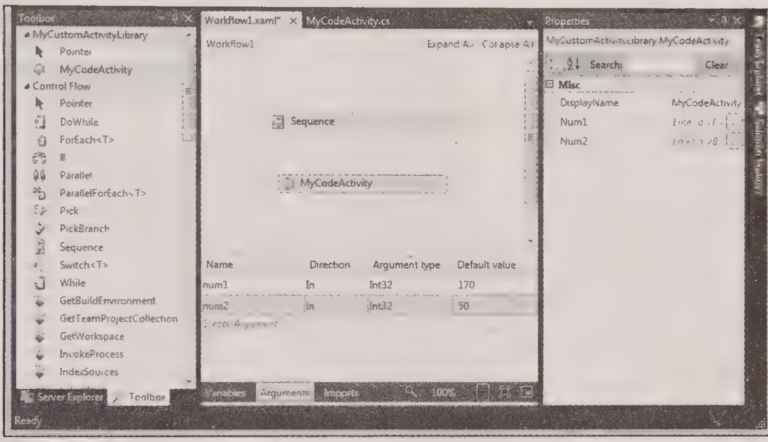


Fig.C#-12.22

27 *Set* the properties of the **MyCodeActivity** custom activity in the **Properties** window, as shown in Fig.C#-12.23:

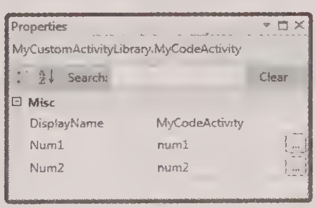


Fig.C#-12.23

28 *Open* the **Program.cs** file, which contains the code shown in Listing 12.3:

Listing 12.3: Displaying the Code of the **Program.cs** File

```
using System;
using System.Linq;
using System.Activities;
using System.Activities.Statements;
```

```
namespace CustomLibraryTestApplication {
    class Program {
        static void Main(string[] args) {
            WorkflowInvoker.Invoke(new Workflow1());
        }
    }
}
```

In Listing 12.3, the **Invoke()** method of the **WorkflowInvoker** class is used to invoke the **CustomLibraryTestApplication** Workflow application.

Now, set the **CustomLibraryTestApplication** Workflow application as the start up project.

29 *Right-click* the **CustomLibraryTestApplication** Workflow application in **Solution Explorer** and *select* the **Set as StartUp Project** option from the context menu that appears, as shown in Fig.C#-12.24:

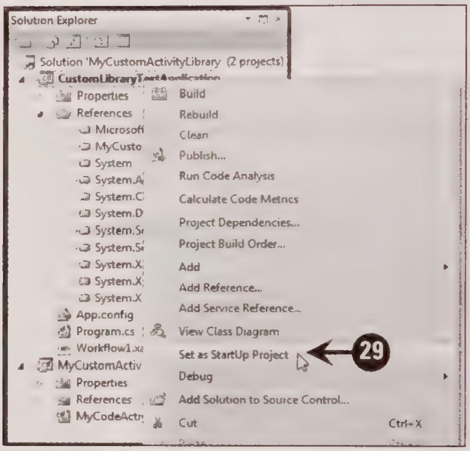


Fig.C#-12.24

30 *Press* the **F5** key to run the **CustomLibraryTestApplication** Workflow application.

The output appears, as shown in Fig.C#-12.25:

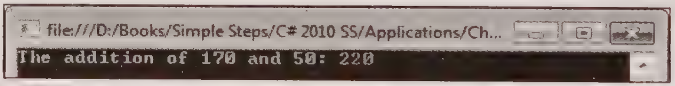


Fig.C#-12.25

In Fig.C#-12.25, the result of addition of the input arguments **num1** and **num2** with default values **170** and **50** respectively is displayed.

Now, let's summarize the key points learned in this chapter.

Summary

In this chapter, you have learned about:

- The design principles of Workflow applications
- Components that constitute WF
- The improvements made to WF
- Creation of a simple WF application and implementation of conditions in a WF application
- Creation of a Workflow activity library and using this library with other applications

Chapter 13

Working with Web and WCF Services

In this Chapter:

- Exploring the New Features of WCF 4.0
- Introducing Cloud Services
- Creating and Using a Web Service
- Creating and Using a WCF Service

A Web Service may be defined as an independent and self-sustained unit of a software application hosted on Internet. It implements specific functionalities to execute the business logic of an application. Suppose you want to host a Web site on Internet that calculates simple interest. In this situation, you can define a separate code for calculating simple interest at the client end in a Web Service and implement the Web Service in your Web site.

A Web Service allows a Web site to communicate with other Web sites irrespective of the programming languages in which they are created. Similarly, it allows Windows Forms applications created in various programming languages to communicate with other Windows Forms applications. As a Web Service complies with common industry standards, it can be accessed by any application, regardless of the software and hardware platforms on which the application is running. The standards to which a Web Service must comply are as follows:

- **Simple Object Access Protocol (SOAP):** Allows Web Services to communicate with other Web applications over the Internet. SOAP is a protocol that specifies a format for Extensible Markup Language (XML) messages, which is used by Web applications to request for and receive a Web Service from a Web server. These messages are also called SOAP messages.
- **Web Services Description Language (WSDL):** Refers to an XML-based language that defines Web Services. Every Web Service consists of a corresponding WSDL document that specifies the location of a Web Service and lists all the services that it can provide.

Earlier, it was difficult to create the applications that were used to communicate between a client and a server as a number of technologies, such as ASP.NET Web Service (ASMX Web Service), Web Service Enhancements (WSE), and Microsoft Message Queuing (MSMQ) were used to perform this task. Some of the services provided individually by these technologies are as follows:

- **ASMX Web Service:** Allows sharing of information and enables the information to be accessed from any platform
- **.NET Remoting service:** Allows transferring of data from a client to a server on the Windows operating system
- **MSMQ service:** Allows queuing of messages that helps in sending and receiving messages even when the server is disconnected

The Windows Communication Foundation (WCF) technology was introduced by Microsoft to simplify the process of developing a distributed application. It provides a unified programming model by bringing together the functionalities of various technologies, such as SOAP, WSDL, ASMX Web Service, WSE, and MSMQ. Using WCF a developer can create a distributed application without mastering various technologies.

WCF is a set of .NET technologies that act as a platform for creating and distributing connected applications, the applications that are connected to a database. You can develop and deploy services on Windows using WCF, which provides a unified framework for rapidly building service-oriented applications. These service oriented applications help to build secure and reliable applications through the simplified model of WCF.

Now, let's learn about the new features in WCF 4.0.

Exploring the New Features of WCF 4.0

Built on the features provided by WCF 3.5 SP1, the WCF 4.0 brings a number of significant changes and improvements. The major enhancements have been made in areas of configuration, tracing and diagnostics, serialization, and messaging. Following are the features of WCF 4.0:

- **Simplification in Configuration:** Refers to the simplifications made in WCF configuration section that includes support for default endpoints, binding, and behavior configurations. WCF 4.0 has a default configuration model; therefore you do not need to define configuration for WCF application. In case you do not define configuration for your WCF application, then the WCF runtime automatically defines some endpoints and default binding/behavior configuration for your WCF application.

- **Default Endpoints:** Refers to the automatic application of default endpoints during the configuration of WCF application. An endpoint basically includes an address, a binding, and a contract. All WCF applications are defined with one or more endpoints; however, while creating WCF 4.0 applications, there is no need to define endpoints.
- **Simplified IIS/ASP.NET Hosting:** Implies that there is no need to write interface definition as you can directly define a class and add ServiceContract and Service Operation with Operation Contract to the class. The following code snippet shows addition of two numbers in a WCF application:

```
[ServiceContract]
 public class Addition
 {
 [OperationContract]
 public double AddNumbers(double a, double b)
 {
    return a + b;
 }
}
```

- **Support for integration of WCF and WF:** Refers to the support that has been added for the integration of WCF and WF. This integration is intended to help implement declarative long-running workflow services. In this way, the developers can use the best features of both WCF and WF technologies.
- **Performance tracing based on Event Tracing for Windows (ETW):** Refers to the improvements in tracing as WCF 4.0 trace events are based on ETW. The performance tracing helps to integrate WCF with other related technologies such as WF and also enhances the tracing performance of WCF applications.

Now, let's learn about the cloud services in the following section.

Introducing Cloud Services

Microsoft introduced cloud services in .NET Framework 4.0. These cloud services are modeled on the lines of cloud computing, which is an Internet-based computing facility. Using the cloud computing facility, resources (such as CPU and memory) software, data, and information can be shared amongst the computer systems and other devices, such as mobile phones with Internet. In other words, cloud computing provides hardware-independence to application developers.

A cloud is used to refer to Web-based applications, services that are hosted on Web, centralized data centre, or an environment where one can build and run scalable applications. Cloud services can be used to develop the Web applications that work well even if there is a sudden increase or decrease in the Web traffic. Using cloud services, previously built desktop or Web applications can take advantage of cloud computing facility.

The newly unveiled Microsoft's cloud platform is named as Windows Azure. It provides an execution environment wherein .NET applications can be run and data can be stored in Microsoft's data repositories that are spread across the world. Using Windows Azure, Microsoft expects to bestow cloud computing facilities to users. Developers can access any of the technologies such as WCF and WF any time and anywhere using Windows Azure. Moreover, Windows Azure allows developers to concentrate on the development aspect of applications rather than worrying about infrastructure requirements. Now, let's learn to use a Web Service in the following section.

Creating and Using a Web Service

You need to create an ASP.NET Empty Web Application template found in the list of templates available in the New Project dialog box to create a Web service application in Visual Studio 2010. Let's perform the following steps to create a Web Service:

1. *Open* **Microsoft Visual Studio 2010 from Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010.**
2. *Select* **File→New→Project** from menu bar. The **New Project** dialog box appears (Fig.C#-13.1).

3. *Select* **Visual C#→Web** under the **Installed Templates** pane and **ASP.NET Empty Web Application** from the middle pane in the **New Project** dialog box (Fig.C#-13.1).
4. *Enter* the name of the Web application as **AddNumbers** in the **Name** text box and select the desired location in the **Location** combo box to save the application (Fig.C#-13.1).
5. *Click* the **OK** button to create the **AddNumbers** Web application, as shown in Fig.C#-13.1:

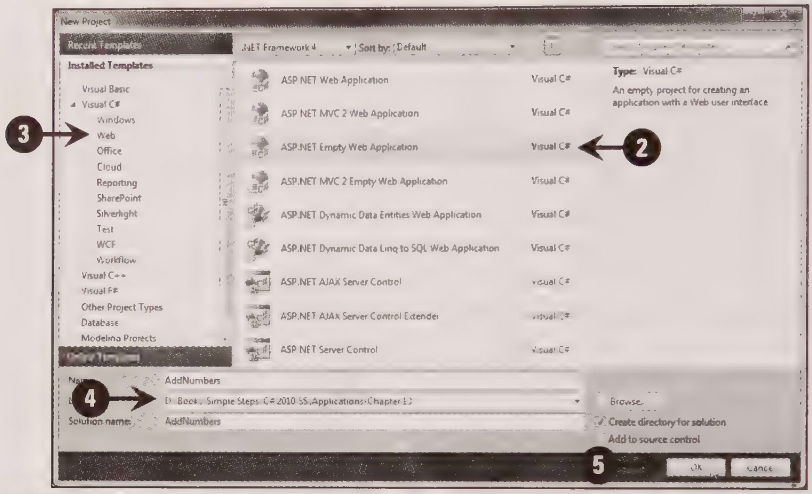


Fig.C#-13.1

An empty Web application is created.

6. *Right-click* **AddNumbers→Add→New Item** in **Solution Explorer**.

   The **Add New Item** dialog box opens (Fig.C#-13.2).
7. *Select* the **Web Service** option from the middle pane (Fig.C#-13.2).
8. *Enter* a name for **Web Service**. In this case, we have entered the name as **MyWebService**.asmx (Fig.C#-13.2).
9. *Click* the **Add** button to add the **MyWebService** Web Service to your application, as shown in Fig.C#-13.2:

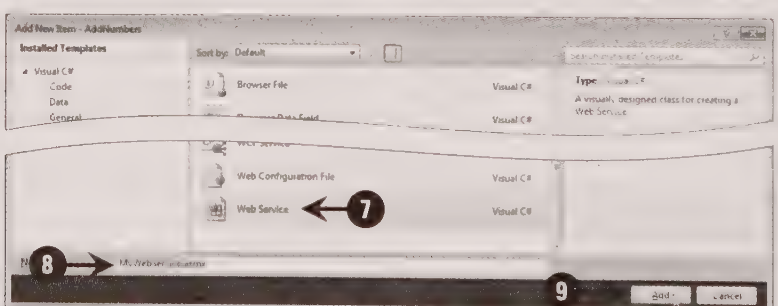


Fig.C#-13.2

You can see the default files of the Web Service in **Solution Explorer**, as shown in Fig.C#-13.3:

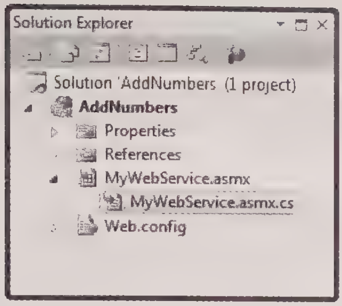


Fig.C#-13.3

10 *Add* a Web method, **Add()** to the code-behind file, **MyWebService.asmx.cs** of the **MyWebService** Web Service, as given in Listing 13.1:

Listing 13.1: Showing the Code to be Added to the **AddNumbers** Web Service

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

namespace AddNumbers
{
    /// <summary>
    /// Summary description for MyWebService
    /// </summary>
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
    // To allow this Web Service to be called from script, using ASP.NET AJAX,
        uncomment the
     //following line.
    // [System.Web.Script.Services.ScriptService]
    public class MyWebService : System.Web.Services.WebService
    {

        [WebMethod]
        public Decimal Add (Decimal a, Decimal b)
        {
            Decimal dc;
            dc = a + b;
            return dc;
        }
    }
}
```

In the Listing 13.1, the **Add()** method adds two **Decimal** numbers and then returns a **Decimal** number as output to the calling method.

11 *Press* the **F5** key to execute the **MyWebService** Web Service. A Web page appears (Fig.C#-13.4).

12 *Click* the **Add** hyperlink on the Web page, as shown in Fig.C#-13.4:

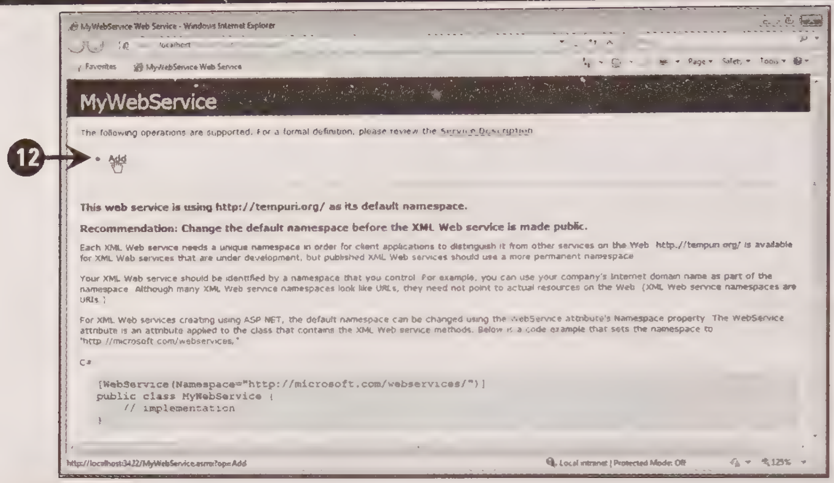


Fig.C#-13.4

The output appears (Fig.C#-13.5).

13 *Enter* values for **a** and **b** parameters (Fig.C#-13.5).

14 *Click* the **Invoke** button, as shown in Fig.C#-13.5:

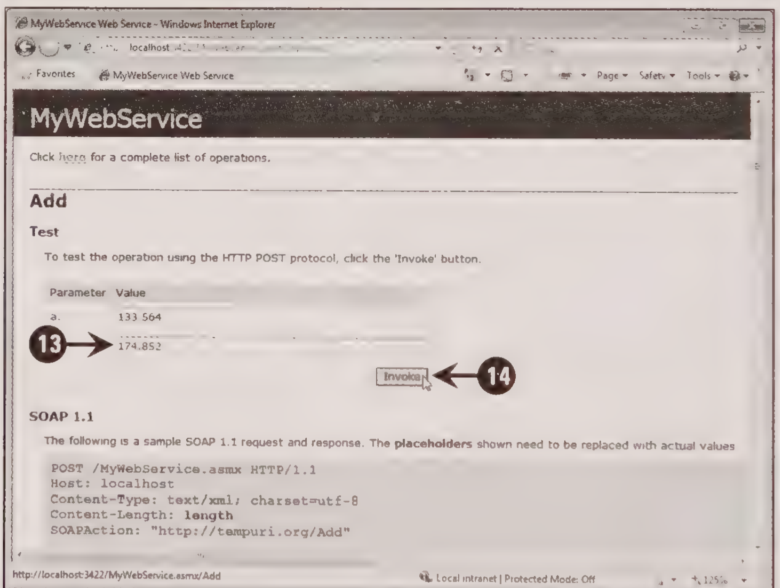


Fig.C#-13.5

A new Web page appears, displaying the result of addition of two values assigned to **a** and **b** parameters, as shown in Fig.C#-13.6:

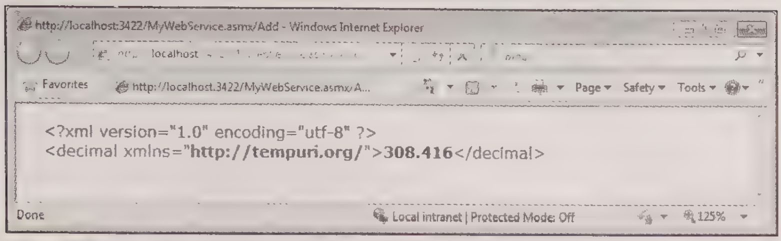


Fig.C#-13.6

Now to access the **MyWebService** Web Service, you need to first add its Web reference to your application. Next, use a proxy class so that your application can access the **MyWebService** Web Service. A proxy class is a local copy of the class that represents a Web Service. You can generate a proxy class for a Web Service by adding a Web reference to a Windows application. In this example, we provide two numbers as input for addition.

Note

In this case, we are creating a Windows Forms Application to access a Web Service application. The Windows Forms Application and Web Service application are both being created on the same system. Therefore, for the Windows Forms application to be able to access the Web Service application properly, you need to keep the instance of Visual Studio 2010 running the Web Service application open.

Now, let's perform the following steps to create a Windows Forms application named **WebServiceExample** to access the **MyWebService** Web Service.

1. *Open* **Visual Studio 2010** and create a Windows Forms Application project, **WebServiceExample**.
2. *Set* the **Text** property of the Windows form to **Web Service Example**.
3. *Add* two labels, two text boxes and a button to the Windows form and change their **Text** properties, as shown in Fig.C#-13.7:

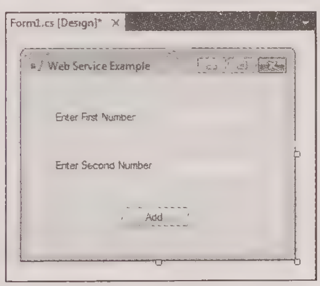


Fig.C#-13.7

4. *Add* a Web reference of the **MyWebService** Web Service to the **WebServiceExample** application by right-clicking the **WebServiceExample** project in **Solution Explorer** and selecting the **Add Service Reference** option from context menu, as shown in Fig.C#-13.8:

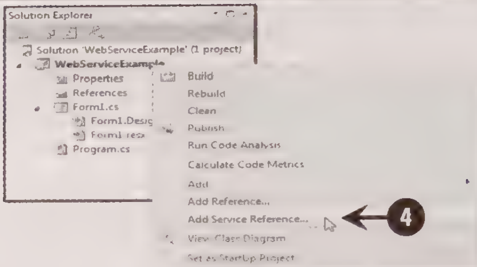


Fig.C#-13.8

The **Add Service Reference** dialog box appears (Fig.C#-13.9).

5 *Click* the **Advanced** button, as shown in Fig.C#-13.9:

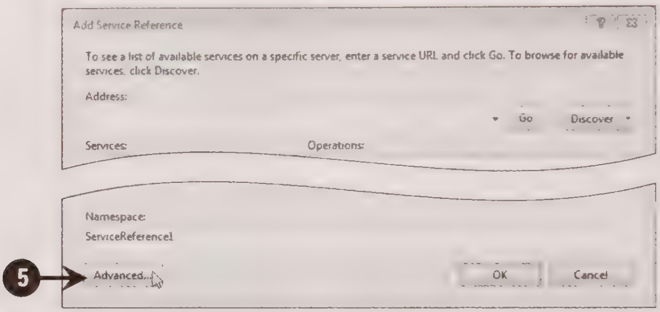


Fig.C#-13.9

The **Service Reference Settings** dialog box appears (Fig.C#-13.10).

6 *Click* the **Add Web Reference** button, as shown in Fig.C#-13.10:

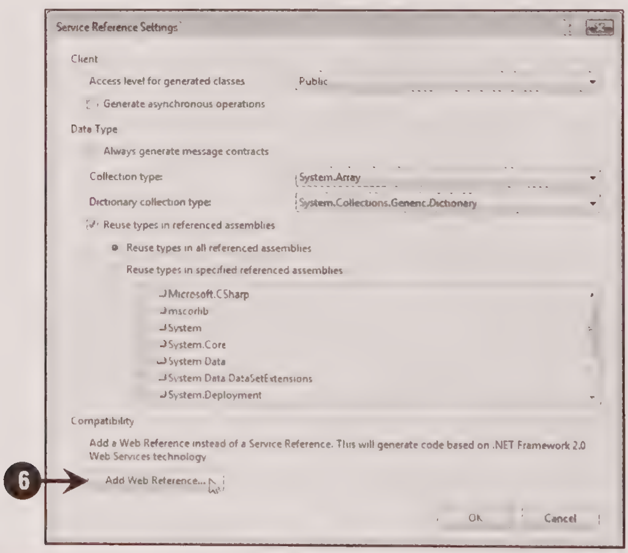


Fig.C#-13.10

The **Add Web Reference** dialog box appears.

7. *Locate* the **MyWebService** Web Service either by specifying its location in the URL text box (the URL should be the URL of the Web Service where the Web Service is hosted) or by selecting an appropriate link under the **Browse** to: option (Fig.C#-13.11)
8. *Click* the **Go** button, as shown in Fig.C#-13.11:

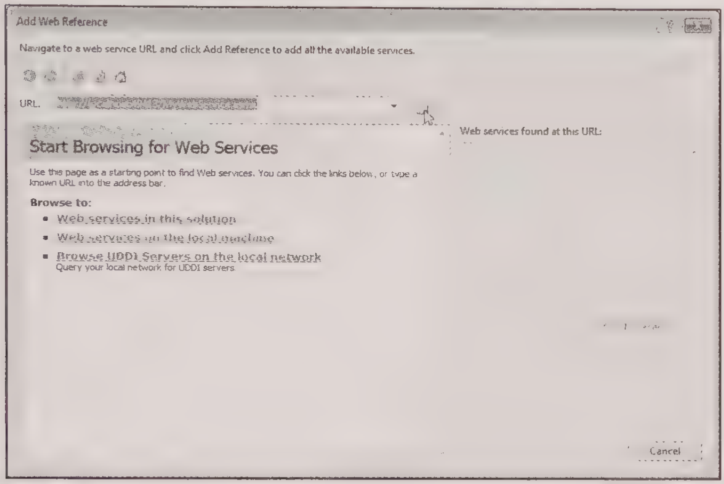


Fig.C#-13.11

The Web Service found at the specified location is displayed (Fig.C#-13.12).

9. *Enter* a name for the Web reference as **AddTwoNumbers** in the **Web reference name** text box (Fig.C#-13.12).
10. *Click* the **Add Reference** button to add the **MyWebService** Web Service to your application, as shown in Fig.C#-13.12:

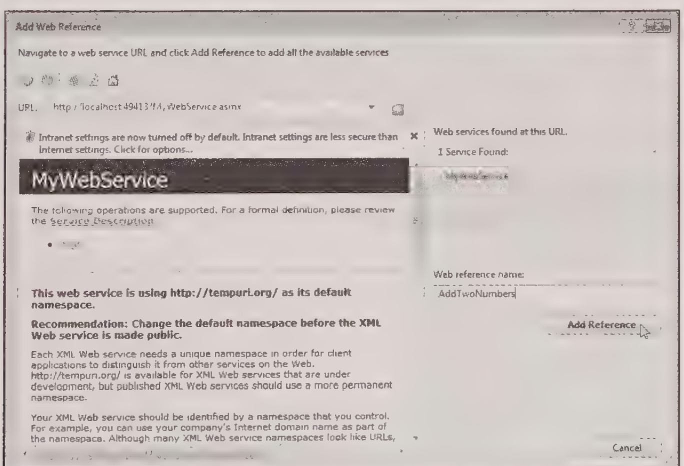


Fig.C#-13.12

After adding the Web reference to the **WebServiceExample** application, you need to create an instance of the Web reference. You can access the Web method of a Web Service using this instance.

11 *Add* the highlighted code shown in Listing 13.2 in the **Form1.cs** file of the **WebServiceExample** application at the **Click** event of the **Add** button to access the **Add()** Web method:

Listing 13.2: Showing the Code to Access a Web Service

```
private void button1_Click(object sender, EventArgs e)
        {
            Decimal a, b, result;
            AddTwoNumbers.MyWebService ad = new AddTwoNumbers.MyWebService();
            a = Convert.ToDecimal(textBox1.Text);
            b = Convert.ToDecimal(textBox2.Text);
            result = ad.Add(a, b);
            MessageBox.Show("The addition of the two numbers is " + result.ToString());
        }
```

In Listing 13.2, the **AddTwoNumbers** namespace has been used to access the **MyWebService** Web Service. The **Add()** method of the **MyWebService** Web Service takes two numbers as parameters for addition. The result of the addition of the two numbers is stored in the result variable, which is displayed in a message box.

12 *Press* the **F5** key to run the application. The output appears (Fig.C#-13.13).

13 *Enter* two numbers in two text boxes and click the **Add** button. A message box appears, as shown in Fig.C#-13.13:

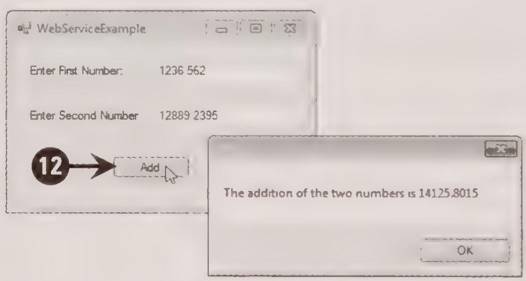


Fig.C#-13.13

14 *Click* the **OK** button to close the message box.

You have learned about Web Services. Now, let's learn to create a WCF service and a client to use the WCF services in the following section.

Creating and Using a WCF Service

In Visual Studio 2010 it is an easy task to create a WCF service as Visual Studio 2010 has an in-built support for WCF. However, in versions prior to Visual Studio 2008, you needed to install the components of WCF to create WCF services. Now, let's create WCF service using the template provided in Visual Studio 2010 by performing the following steps:

1 *Open* **Microsoft Visual Studio 2010**.

2 *Select* **File→New→Project** from menu bar. The **New Project** dialog box appears.

3 *Select* **Visual C#** under the **Installed Templates** pane and **WCF Service Application** from the middle pane in the **New Project** dialog box.

4. *Enter* a name as **WCFService** for the WCF Service application and select the desired location in the **Location** combo box to save the application.

5. *Click* the **OK** button to create the **WCFService** WCF Service.

   The default files of the WCF Service appear in **Solution Explorer**, as shown in Fig.C#-13.14:

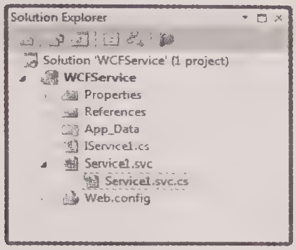


Fig.C#-13.14

You can build a WCF Service by performing two steps; first, you need to create a service contract. Secondly, you need to create a data contract. A service contract specifies the rules for interfaces along with their corresponding messages and a data contract specifies the rules for the data that is exchanged through input and output operation.

6. *Add* the highlighted code shown in Listing 13.3 in the **IService1.cs** file of the **WCFService** service:

Listing 13.3: Showing the Code for the **IService1.cs** File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Web;
using System.Text;

namespace WCFService
{
    // NOTE: You can use the "Rename" command on the "Refactor" menu to change
         the interface
    //name "IService1" in both code and config file together.
    [ServiceContract]
    public interface IService1
    {

        [OperationContract]
        string GetData(string value);

        [OperationContract]
        string GetDataUsingDataContract(UserInfo users);

        // TODO: Add your service operations here
    }


    // Use a data contract as illustrated in the sample below to add composite
         types to service
     //operations.
    [DataContract]
    public class UserInfo
```

```
    {
        string fName;
        string lName;

        [DataMember]
        public string FirstName
        {
            get { return fName; }
            set { fName = value; }
        }
        [DataMember]
        public string LastName
        {
            get { return lName; }
            set { lName = value; }
        }
    }
}
```

In Listing 13.3, the **IService1** interface is labeled with the [**ServiceContract**] attribute and explicitly mark the **IService1** interface to carry contract metadata for WCF. The methods within the **IService1** interfac labeled with the [**OperationContract**] attribute are marked as public methods in WCF. A data contract i defined for the **UserInfo** class and its members by using the [**DataContract**] attribute and th [**DataMember**] attribute respectively. Service contracts define the behavior of a service whereas dat contracts define the information that flows across service boundaries and are handled with additional logi on the provider and consumer side.

7 *Add* the highlighted code given in Listing 13.4 in the **Service1.svc.cs** file of the **WCFService** service:

Listing 13.4: Showing the Code for the **Service1.svc.cs** File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Web;
using System.Text;

namespace WCFService
{
    // NOTE: You can use the "Rename" command on the "Refactor" menu to change the
          class name
    //"Service1" in code, svc and config file together.
    public class Service1 : IService1
    {
        public string GetData(string Myvalue)
        {
            return "Hello: " + Myvalue;
        }

        public string GetDataUsingDataContract(UserInfo dcValue)
        {
            return "Hello: " + dcValue.FirstName + " " + dcValue.LastName;
        }
    }
}
```

8 *Press* the **F5** key to execute the **WCFService** service. A **WCF Test Client** window appears, as shown i Fig.C#-13.15:

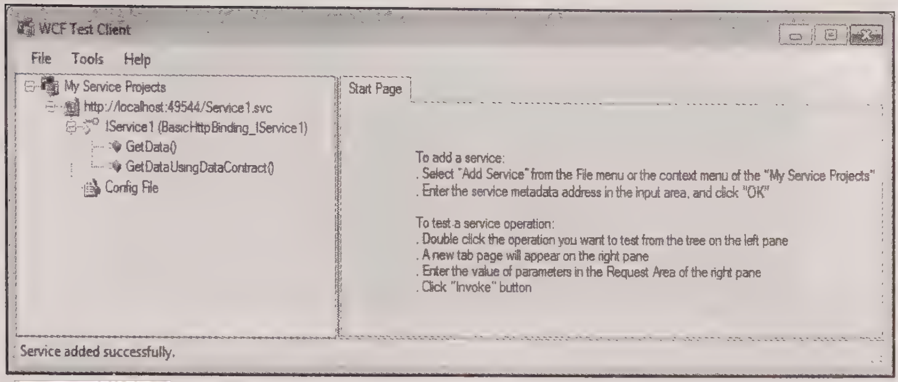


Fig.C#-13.15

9. *Copy* the URL of WCF Service application and paste it in your Web browser. The output appears, as shown in Fig.C#-13.16:

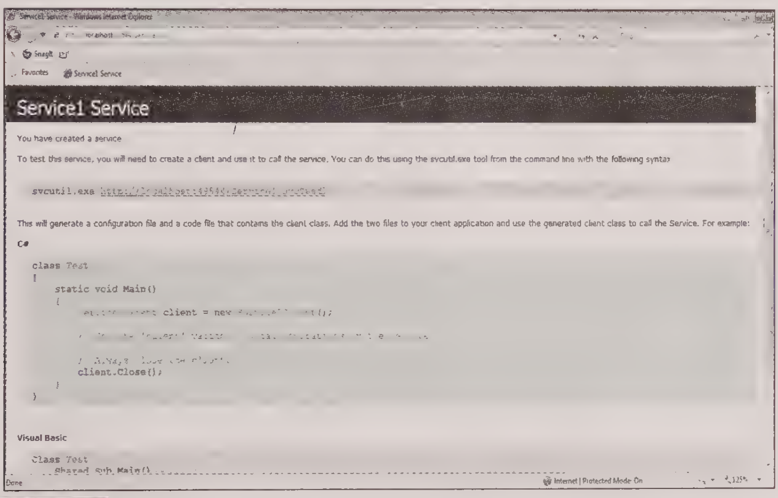


Fig.C#-13.16

Now, a client application is required to use this WCF Service. You have to generate the proxy classes that help the WCF client application to call the WCF Service before creating the WCF client. Execute the command given in the following code snippet in Visual Studio 2010 Command Prompt to generate proxy classes:

```
svcutil.exe /language:cs /out:Service1.cs /config:app.config
    http://localhost:49544/Service1.svc?wsdl
```

In the preceding code snippet, the **svcutil.exe** command generates a configuration file (**app.config**) and a code file (**Service1.cs**) containing the proxy class.

Note

Execute the svcutil.exe command in the Command Prompt at the location where the WCF Service is created.

The output after executing the command given in the preceding code snippet is shown in Fig.C#-13.17:

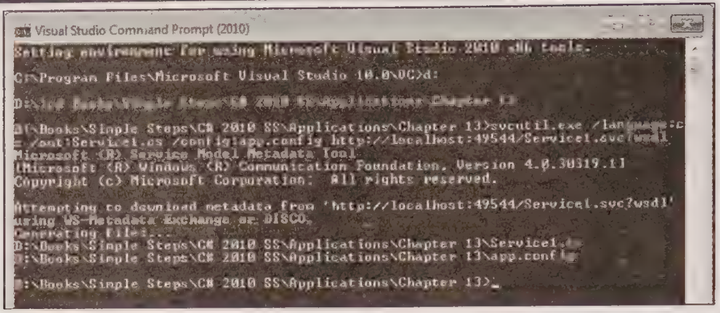


Fig.C#-13.17

Fig.C#-13.17 shows the **Service1.cs** and **app.config** files that are generated after executing the **svcutil.exe** command.

Note

The port number, which is 49544, may vary according to your system.

Now, let's perform the following steps to create a WCF client application:

1. *Open* **Visual Studio 2010** and create a **Windows Forms** application, **WCFClient**.
2. *Set* the **Text** property of the form to **WCF Client**.
3. *Right-click* the name of the project in Solution Explorer and select the **Add Service Reference** option from the context menu to add the reference of the **WCFService** WCF Service after creating the **WCFClient** application.
4. *Enter* the address of the **WCFService** WCF Service and click the **Go** button in the **Add Service Reference** dialog box. The list of services is displayed (Fig.C#-13.18).
5. *Click* the **OK** button to add service reference, as shown in Fig.C#-13.18:

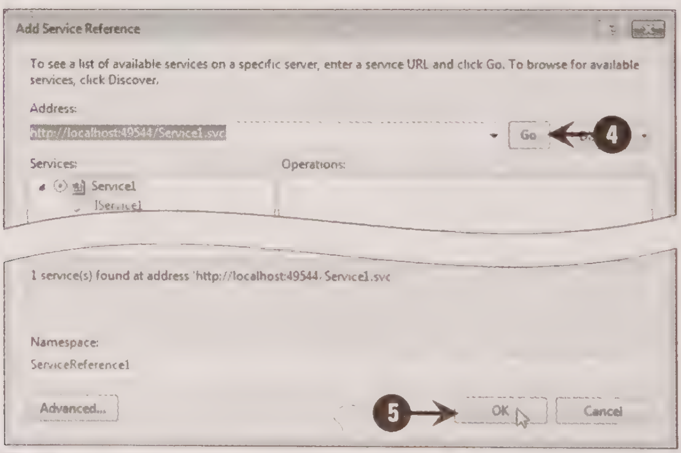


Fig.C#-13.18

6. *Add* three **Label** controls, two **TextBox** controls, and a **Button** control and set their **Text** properties, as shown in Fig.C#-13.19:

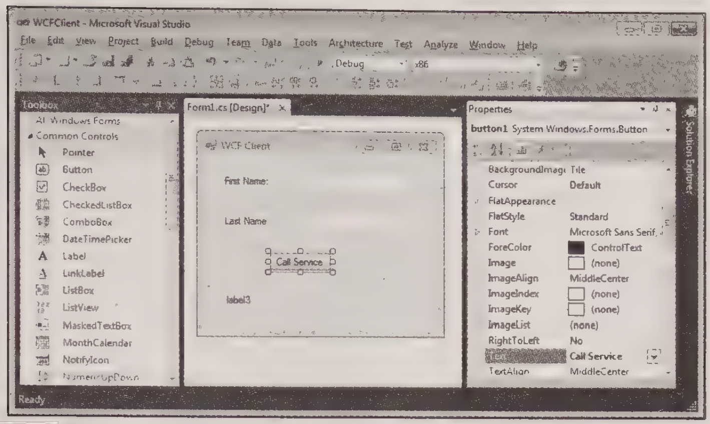


Fig.C#-13.19

7. *Add* the **Service1.cs** file (generated using the **svcutil.exe**) to the **WCFClient** application after adding the controls to the Windows form.
8. *Replace* the original **app.config** file with the **app.config** file (generated using the **svcutil.exe**) in the **WCFClient** application folder.
9. *Add* the highlighted code given in Listing 13.5 on the **Click** event of the **Call Service** button in the **Form1.cs** file:

Listing 13.5: Calling the **WCFService** Service

```
private void button1_Click(object sender, EventArgs e)
{
    Service1Client client = new Service1Client();
    WCFService.UserInfo users= new WCFService.UserInfo ();
    users.FirstName = textBox1.Text;
    users.LastName = textBox2.Text;
    label3.Text = client.GetDataUsingDataContract(users);
    client.Close();
}
```

10. *Press* the **F5** key to execute the **WCFClient** application (Fig.C#-13.20).
11. *Enter* the first name and last name in the respective text boxes (Fig.C#-13.20).
12. *Click* the **Call Service** button to call the **WCF Service**. The output after calling the WCF Service is shown in Fig.C#-13.20:

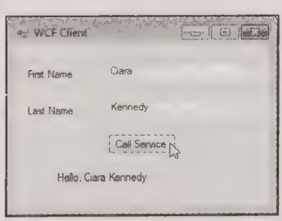


Fig.C#-13.20

In Fig.C#-13.20, when the **Call Service** button is clicked, the **WCFService** service is called, which returns a string **Hello**: **<FirstName> <LastName>**.

Next, let's recapitulate the main points that you have learned in this chapter.

Summary

In this chapter, you have learned about:

- The new features in WCF 4.0, Web Services, and WCF Services
- Cloud services and the purpose of their usage
- Creation and usage of Web Services and WCF Services in Visual Studio 2010

Chapter 14

Deployment of C# 2010 Applications

In this Chapter:

- Deploying Applications Using Windows Installer
- Deploying Applications Using ClickOnce

Imagine a scenario where you have built a software on your computer and you want to use the same software on another computer. You can do this by deploying the same software on another computer.

Deployment is the process that makes software available for use by just installing it on a computer. You need to create setup files and then install the software on the computer of a user.

Visual C# 2010 applications are designed to be deployed and installed with the Windows installer program, which uses Microsoft Installer (.msi) files. In addition, Visual C# 2010 also enables you to deploy your applications by using another technique called the **ClickOnce** deployment.

In this chapter, learn to create .msi files for applications. Next, you learn to install an application, where you just need to copy and execute the .msi file on the user machine and then Installer Wizard performs a series of steps to install an application. Let's first learn to deploy an application using Windows installer.

Deploying Applications Using Windows Installer

Windows Installer allows you to deploy a Windows application by creating a Windows Installer Package. The installer package has .msi extension and it contains the Windows application, dependent files, and registry entries. After creating an application, you need to transfer the .msi file to the target machine and then double-click the .msi file to install it. Ensure that the target machine supports Windows Installer and .NET Framework so that your application can function properly. You can create .msi installer files with Setup and Deployment projects in Visual C# 2010.

Let's perform the following steps to create and deploy an application named **FillColor**:

1. *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010** to open Visual Studio 2010 IDE (Integrated Development Environment).
2. *Select* **File→New→Project** to open the **New Project** dialog box In the Visual Studio 2010 IDE.
3. *Select* **Visual C#→Windows** in the **Installed Templates** pane and the **Windows Forms Application** option in the middle pane of the **New Project** dialog box.
4. *Enter* the name as **FillColor** in the **Name** text box and an appropriate location in the **Location** combo box to save the application.
5. *Click* the **OK** button. The **Design** view of a blank Windows Form appears.
6. *Select* **Form1** in the **Design** view and *set* the **Text** property of **Form1** as **Fill Color**.
7. *Drag* **PictureBox**, **Button**, **ProgressBar**, **Timer**, and **ColorDialog** controls from the **Toolbox** and *drop* on the Windows Form.
8. *Set* the **Text** property of the button as **Pick Color**.
9. *Upload* an image in picture box through its **Image** property and *set* the **BackColor** property of picture box to **ButtonHighlight**.

Note

To know more about working with PictureBox control refer to the Using the PictureBox control section of Chapter 5, Programming with Windows Forms Controls.

10. *Double-click* the **Pick Color** button and *add* the following code snippet on its **Click** event in the **Form1.cs** file:

```
if (colorDialog1.ShowDialog() == DialogResult.OK) {
  progressBar1.Value = 0;
  timer1.Enabled = true;
}
```

The preceding code snippet shows the code that opens a **Color** dialog box when a user clicks the **Pick Color** button. The **Timer** control is enabled as soon as the user selects a color from the **Color** dialog box and clicks the **OK** button.

11 *Double-click* the **Timer** control again in the Design view and *add* the following code snippet on the **Tick** event of the timer:

```
progressBar1.Value += 10;
if (progressBar1.Value == progressBar1.Maximum) {
  pictureBox1.BackColor = colorDialog1.Color;
  timer1.Enabled = false;
}
```

The preceding code snippet fires the **Tick** event of the timer when the timer is enabled. In **Tick** event handler of the timer, we are incrementing the progress bar value by 10. When the progress bar value reaches the maximum value, which is 100, the selected color is loaded in the picture box and the timer is disabled.

12 *Press* the **F5** key to execute the application. The output appears.

13 *Click* the **Pick Color** button to open the **Color** dialog box.

14 *Select* a color from the **Color** dialog box and *click* the **OK** button.

The **FillColor** application is created. Let's learn to deploy this application by using the following steps:

15 *Select* **File→Add→New Project** to open the **Add New Project** dialog box (Fig.C#-14.1).

16 *Select* **Other Project Types→Setup** and **Deployment→Visual Studio Installer** under the **Installed Templates** pane (Fig.C#-14.1).

17 *Select* the **Setup Wizard** option in the middle pane (Fig.C#-14.1).

18 *Enter* a name for the project in the **Name** text box. In our case, we have entered, **FillColorSetup** (Fig.C#-14.1).

19 *Click* the **OK** button, as shown in Fig.C#-14.1:

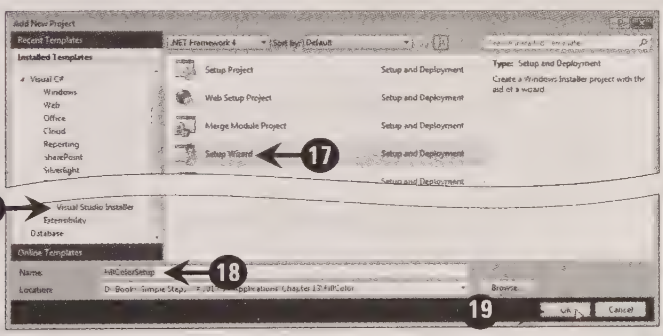


Fig.C#-14.1

The **Setup Wizard** appears and displays the **Welcome to the Setup Project Wizard** page, as shown in Fig.C#-14.2:

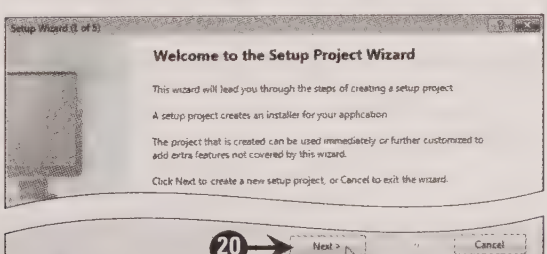


Fig.C#-14.2

20 *Click* the **Next** button to move to the **Choose a project type** page of the **Setup Wizard** (Fig.C#-14.2).

21 *Select* a project type in which you want to deploy the application. In this case, we have selected the **Create a setup for a Windows application** radio button, as shown in Fig.C#-14.3:

22 *Click* the **Next** button to move to the **Choose project outputs to include** page of the **Setup Wizard**, as shown in Fig.C#-14.3:

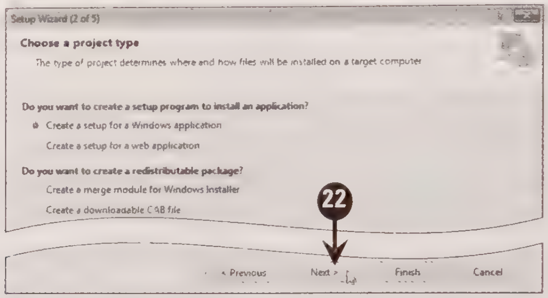


Fig.C#-14.3

In the **Choose project outputs to include** page, you can specify the files that you want to deploy with the application. For example, you can deploy the program itself or the program along with its source code. In our case, we deploy all the files of the **FillColor** application by selecting all items given in the **Choose project outputs to include** page (Fig.C#-14.4).

23 *Click* the **Next** button, as shown in Fig.C#-14.4:

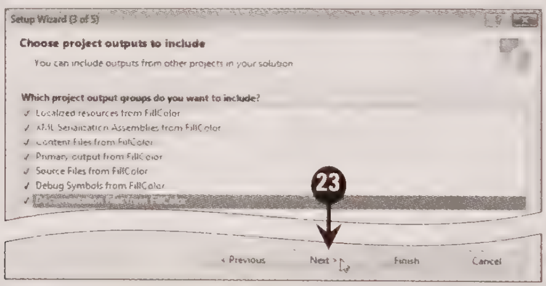


Fig.C#-14.4

The **Choose files to include** page of the **Setup Wizard** appears (Fig.C#-14.5). In the **Choose files to include** page, you can include other files that you want to deploy, such as readme.txt files, and licensing agreements. However, in this case, we are not including such files.

24 *Click* the **Next** button, as shown in Fig.C#-14.5:

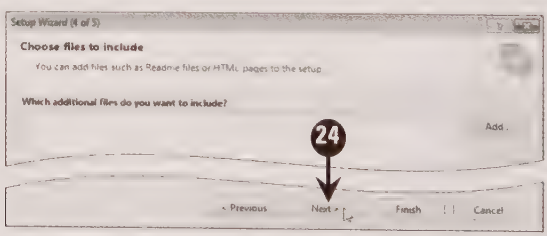


Fig.C#-14.5

The **Create Project** page, which is the final page of the **Setup Wizard,** appears (Fig.C#-14.6).

25 *Click* the **Finish** button in the **Create Project** page to create the installer file, as shown in Fig.C#-14.6:

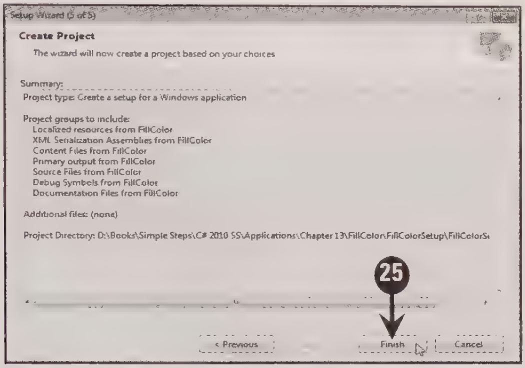


Fig.C#-14.6

The new project, **FillColorSetup** is created.

In the **Properties** window, you can set various properties, such as **ManufactureURL, ProductCode**, and **ProductName** for the **FillColorSetup** project. You can also define the items that you want to create on the user's desktop or user's program menu such as a shortcut for an application.

26 *Select* the **User's Desktop** folder under the **File System on Target Machine** pane (Fig.C#-14.7).

27 *Right-click* the right pane and select the **Create New Shortcut** option from the context menu, as shown in Fig.C#-14.7:

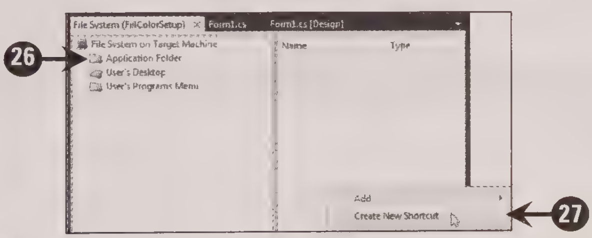


Fig.C#-14.7

The **Select Item in Project** dialog box appears (Fig.C#-14.8).

28 *Double-click* the **Application Folder** folder to view its contents, as shown in Fig.C#-14.8:

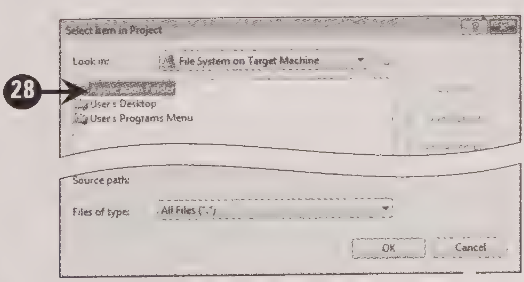


Fig.C#-14.8

29 *Select* the **Primary output from FillColor (Active)** item and then *click* the **OK** button, as shown in Fig.C#-14.9:

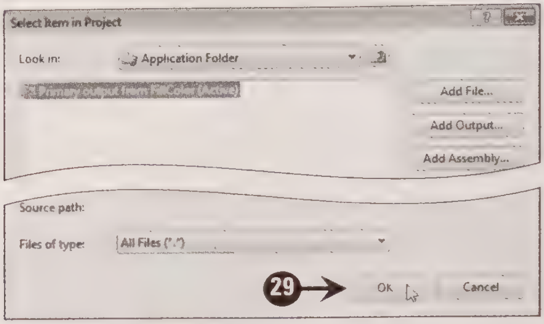


Fig.C#-14.9

The shortcut for the **FillColorSetup** project is created.

30 *Change* the name of the shortcut to be created on the **User's Desktop**. In our case, we have changed the name to **FillColor**, as shown in Fig.C#-14.10:

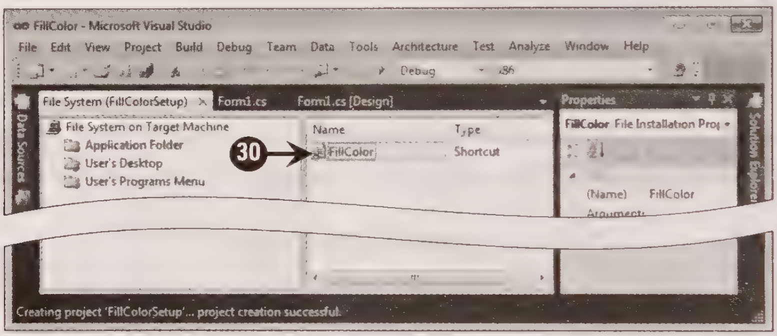


Fig.C#-14.10

31 *Select* the **User's Program Menu** folder under the **File System on Target Machine** pane and then *repeat* steps 27 to 30 to create a shortcut for the program menu.

32 *Select* the **Application Folder** folder under the **File System on Target Machine pane** and set the **AlwaysCreate** property to **True** to specify that for every installation the selected folder is created, as shown in Fig.C#-14.11:

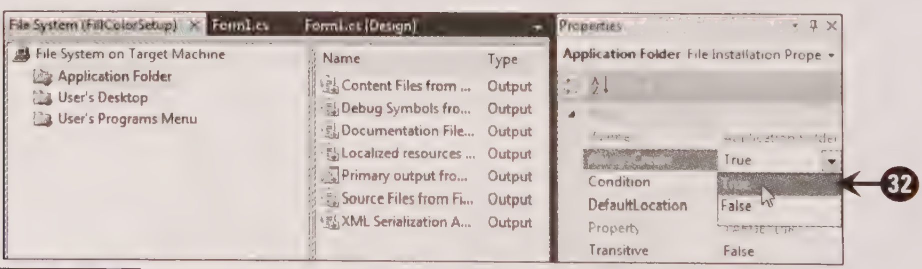


Fig.C#-14.11

33 *Set* the **AlwaysCreate** property to **True** for the **User's Desktop** and **User's Programs Menu** folders as done in case of the **Application Folder** folder.

Note

The AlwaysCreate property specifies whether the selected folder should be created for every installation or not.

34 *Select* **Build→Build FillColorSetup** from the menu bar of the Visual Studio 2010 IDE after the **FillColorSetup** project is created.

The **FillColorSetup.msi** and **Setup.exe** files are created. First you need to locate the **FillColorSetup.msi** file and then copy the **FillColorSetup.msi** file to the target machine to deploy it.

35 *Open* the **Debug** folder of the **FillColorSetup** project to *copy* the **FillColorSetup.msi** file to the target machine, as shown in Fig.C#-14.12:

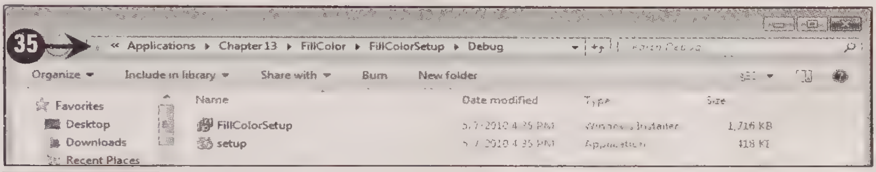


Fig.C#-14.12

36 *Double-click* the **FillColorSetup.msi** file to open the **Welcome to the FillColorSetup Setup Wizard** page of the **FillColorSetup** setup wizard.

37 *Click* the **Next** button to move to the **Select Installation Folder** page of the **FillColorSetup** setup wizard, as shown in Fig.C#-14.13:

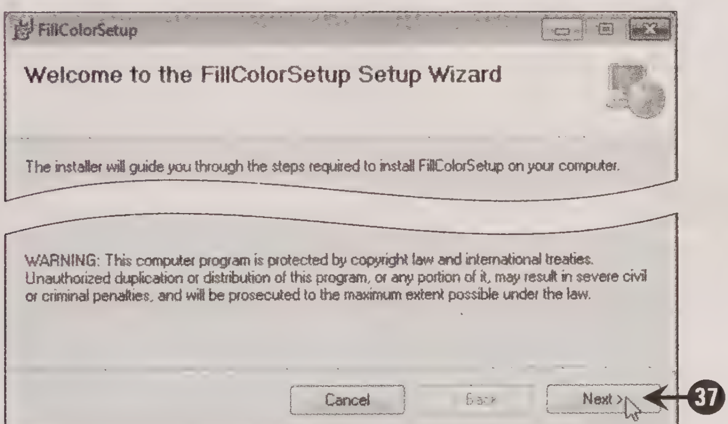


Fig.C#-14.13

The **Select Installation Folder** page appears (Fig.C#-14.14).

38 *Browse* the location of the folder where you want to install the **FillColor** application and *click* the **Next** button to move to the **Confirm Installation** page, as shown in Fig.C#-14.14:

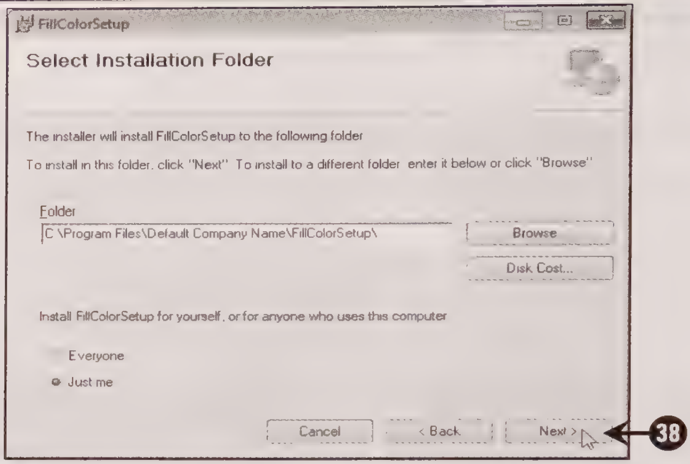


Fig.C#-14.14

The **Confirm Installation** page of the **FillColorSetup** setup wizard appears (Fig.C#-14.15).

39 *Click* the **Next** button to start the installation, as shown in Fig.C#-14.15:

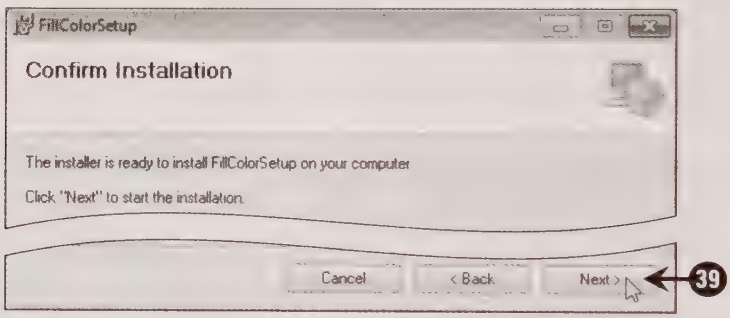


Fig.C#-14.15

The **Installing FillColorSetup** page appears, as shown in Fig.C#-14.16:

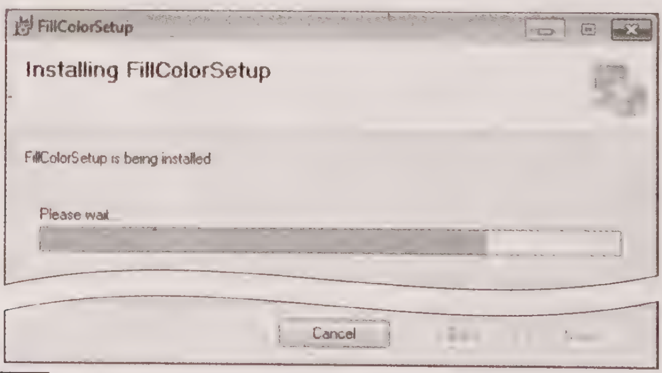


Fig.C#-14.16

After the installation is complete, the **Installation Complete** page appears.

40 *Click* the **Close** button to successfully complete the **FillColorSetup** wizard.

41 *Run* the newly installed **FillColor** application from the location where you have installed it or by selecting **Start→All Programs→FillColor**. The output appears, as shown in Fig.C#-14.17:

Fig.C#-14.17

This completes creating and deploying an application by using Windows Installer. Let's learn to deploy an application by using ClickOnce.

Deploying Applications Using ClickOnce

ClickOnce deployment is used to deploy Windows application on the Web server or network file share. It provides a simplified installation experience to the end user. Using the **ClickOnce** deployment, you can enable the self-update feature of Windows applications that can be installed, updated, and run from a website with minimal user interaction. The self-updating feature enables you to download only the updated portions of the application. The **ClickOnce** deployment method deploys an application by using the **Publish Wizard**. The **Publish Wizard** enables you to publish an application to a Web page, a network file share, or a CD. In other words, **ClickOnce** is an easy technique to install Web applications as compared to the Windows installer deployment technique. Using **ClickOnce** you can install Windows applications through a simple click. Let's perform the following steps to deploy an application using **ClickOnce** (in this case, we deploy the **FillColor** application, which we have created in the previous section):

1 *Click* **Start→All Programs→Microsoft Visual Studio 2010→Microsoft Visual Studio 2010.**

2 *Right-click* **Microsoft Visual Studio 2010** and *select* **Run as administrator** option from the context menu that appears.

3 Create a Windows Forms Application named **FillColour.**

4 *Repeat* steps 6 to 14 of **Deploying Applications by Using Windows Installer** section of this chapter.

5 *Right-click* the **FillColour** project node in **Solution Explorer** and then select the **Publish** option from the context menu, as shown in Fig.C#-14.18:

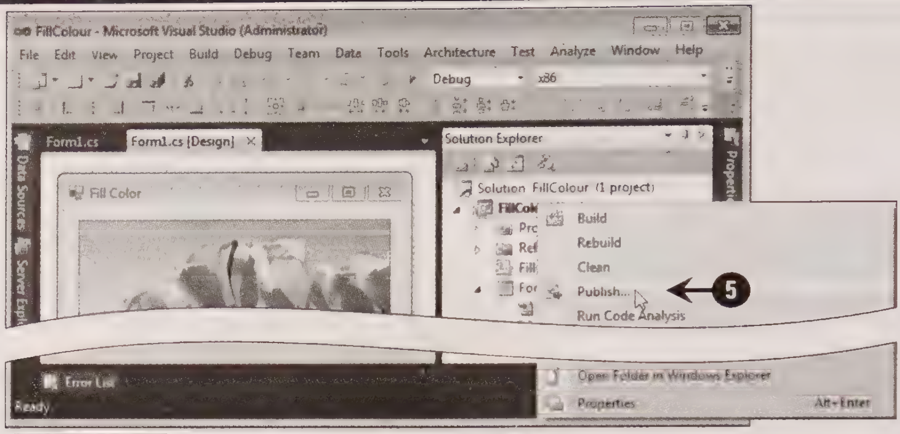


Fig.C#-14.18

The **Publish Wizard** appears and displays the **Where do you want to publish the application** page (Fig.C#-14.19).

6 *Enter* the location where you want to publish your application, such as a website, a network file share or a CD. In our case, we are publishing the **FillColour** application on a website, so we have entered **http://localhost/** in the **Where do you want to publish the application** page and then *click* the **Next** button, as shown in Fig.C#-14.19:

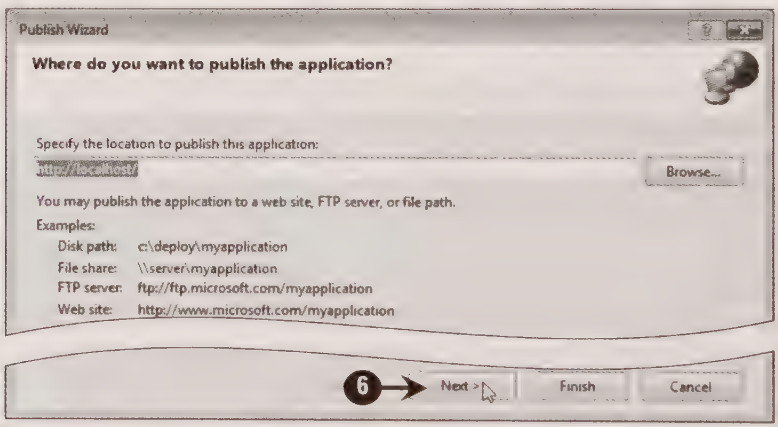


Fig.C#-14.19

The **Will the application be available offline** page of **Publish Wizard** appears (Fig.C#-14.20).

7 *Select* the appropriate option to specify the mode in which you want your application to be available. In this case, we select the **Yes, this application is available online or offline** radio button (Fig.C#-14.20).

8 *Click* the **Next** button, as shown in Fig.C#-14.20:

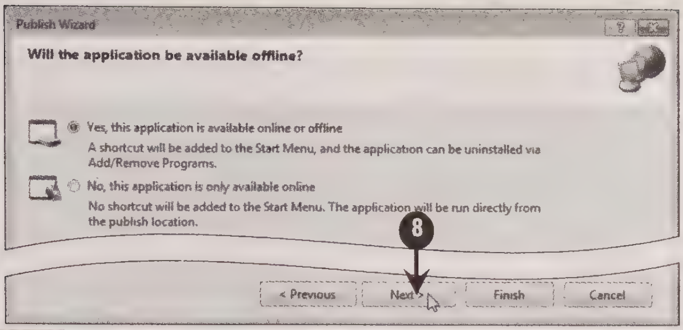


Fig.C#-14.20

The **Ready to Publish!** page displaying the message that the application is ready to publish appears (Fig.C#-14.21).

9 *Click* the **Finish** button to complete the publishing of the application, as shown in Fig.C#-14.21:

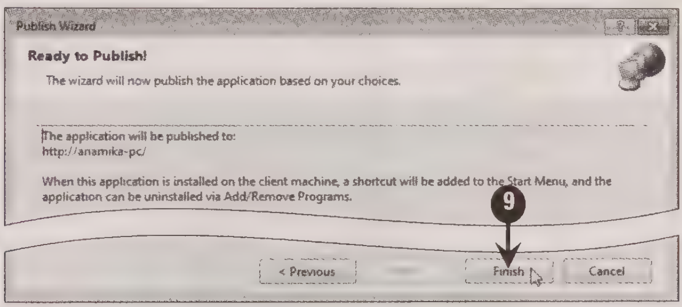


Fig.C#-14.21

The project is built and the application is published to the specified location. The **publish.htm** file opens in the Web browser (Fig.C#-14.22).

10 *Click* the **Install** button, as shown in Fig.C#-14.22:

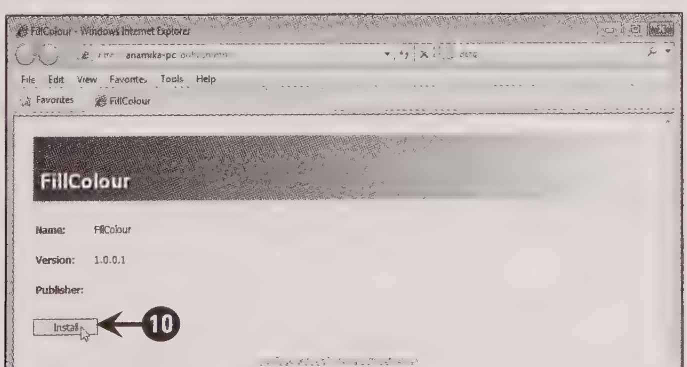


Fig.C#-14.22

The installation of the **FillColour** application starts. A security warning message appears asking whether you want to install the application or not (Fig.C#-14.23).

11 *Click* the **Install** button to install the **FillColour** application, as shown in Fig.C#-14.23:

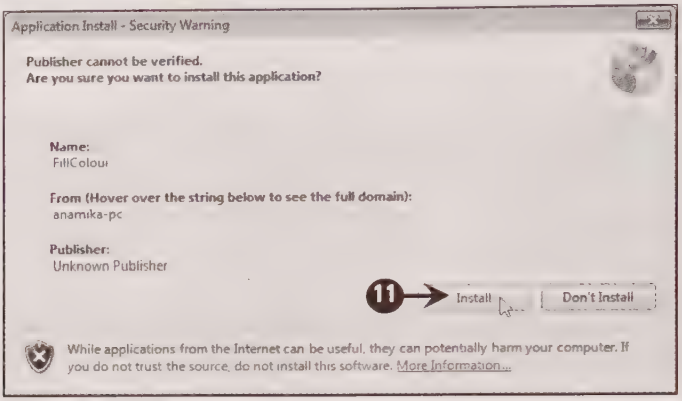


Fig.C#-14.23

After completing the installation, the output appears, as shown in Fig.C#-14.24:

Fig.C#-14.24

After the **ClickOnce** application is installed and executed, an icon of the **ClickOnce** application is displayed on the **Start** menu and an entry is added to the **Programs and Features** option in **Control Panel**. Now, you can open the **ClickOnce** application directly from the **Start** menu and can also uninstall the application by using **Control Panel**.

Internet Information Services (IIS) should be installed on your system to run the **FillColour** application. Perform the following steps to install IIS 7.5 from Control Panel:

1 *Click* **Start→Control Panel**.

The **Control Panel** window appears.

2 *Click* the **Programs** option in the **Control Panel** window.

The **Programs** window appears.

3 *Click* the **Turn Windows Features on or off** option on the right pane of the **Programs** window.

The **Windows Features** window appears (Fig.C#-14.25).

4 *Select* the **Internet Information Services** checkbox (Fig.C#-14.25).

5 *Click* the **OK** button to start installing **IIS** on your system, as shown in Fig.C#-14.25:

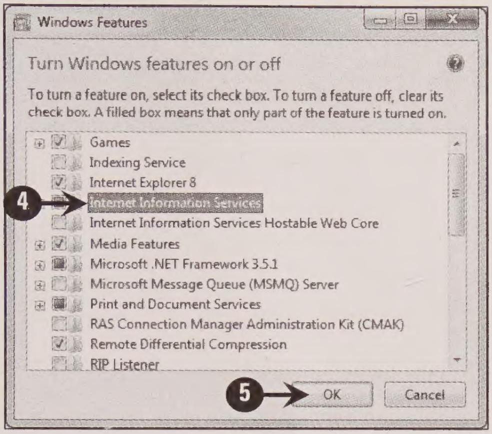


Fig.C#-14.25

A new window appears, showing the installation of IIS, as shown in Fig.C#-14.26:

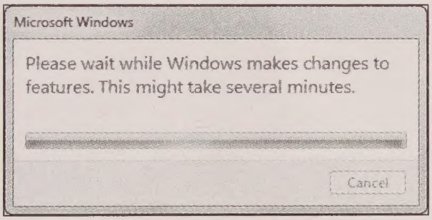


Fig.C#-14.26

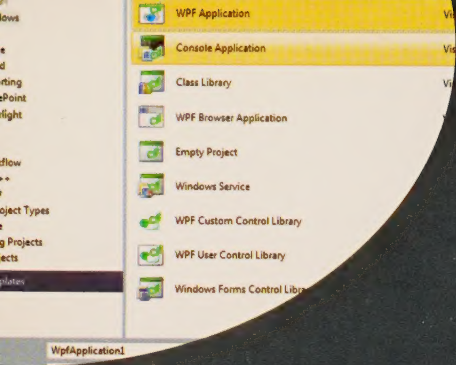
It remains for a few minutes till the installation completes.

Now, let's summarize the different topics that you have learned in this chapter.

Summary

In this chapter, you have learned about:

- Deploying applications using different deployment techniques
- Deploying Windows applications by creating a Windows Installer Package using Windows Installer
- Deploying Windows applications using ClickOnce deployment on the Web server or network file share.



IN SIMPLE STEPS

C# 2010

C# 2010 IN SIMPLE STEPS is a book that helps you to learn C# using Visual Studio 2010. Precision, an easy-to-understand style, real-life examples in support of the concepts, and practical approach in presentation are some of the features that make the book unique in itself. The text in the book is presented in such a way that it is equally helpful to the beginners as well as to the professionals. Apart from the basic concepts, such as Dynamic types lookup, named and optional parameters, parallel LINQ, the book deals with some advanced topics, such as WPF, WCF, and Cloud Computing.

The book covers:

- .NET Framework 4.0 and Visual Studio 2010
- C# 2010 Programming Essentials
- Control Structures and Exception Handling
- Object-Oriented Programming Constructs
- Windows Forms Controls, Menus, Toolbars, and Dialog Controls
- Windows Presentation Foundation and XAML
- ADO.NET and Data Binding
- C# 2010 Delegates, Events, and Lambdas
- Language-Integrated Query and Plinq
- Dynamic Programming
- Windows Workflow Foundation
- Cloud, Web, and WCF Services
- Deployment of C# 2010 Applications

Published by:

dreamtech PRESS

Dreamtech Press
19-A, Ansari Road, Daryaganj,
New Delhi-110002
Tel.: 91-11-23284212, 23243075
Fax: 91-11-23243078
Email: feedback@dreamtechpress.com
http://www.dreamtechpress.com
Blog: http://dreamtechpress.wordpress.com

978-93-5004-032-4

9 789350 040324

KOGENT Learning Solutions Inc.

SPECIAL INDIAN PRICE

Rs. 249/-

International Price $11.99